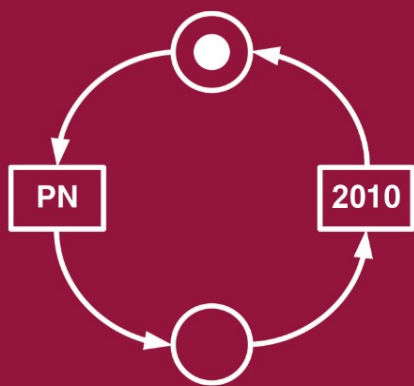


Johan Lilius  
Wojciech Penczek (Eds.)

LNCS 6128

# Applications and Theory of Petri Nets

31st International Conference, PETRI NETS 2010  
Braga, Portugal, June 2010  
Proceedings



*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*

Alfred Kobsa

*University of California, Irvine, CA, 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*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Johan Lilius Wojciech Penczek (Eds.)

# Applications and Theory of Petri Nets

31st International Conference, PETRI NETS 2010  
Braga, Portugal, June 21-25, 2010  
Proceedings

Volume Editors

Johan Lilius

Åbo Akademi University, Department of Information Technologies

Joukahainengatan 3-5, 20520 Turku, Finland

E-mail: johan.lilius@abo.fi

Wojciech Penczek

Polish Academy of Sciences, Institute of Computer Science

Ordonia 21, 01-237 Warsaw, Poland

E-mail: penczek@ipipan.waw.pl

Library of Congress Control Number: 2010928055

CR Subject Classification (1998): F.1.1, D.2, F.3, H.4, D.1, F.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743

ISBN-10 3-642-13674-5 Springer Berlin Heidelberg New York

ISBN-13 978-3-642-13674-0 Springer Berlin Heidelberg New York

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

springer.com

© Springer-Verlag Berlin Heidelberg 2010

Printed in Germany

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

Printed on acid-free paper 06/3180

# Preface

This volume consists of the proceedings of the 31th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS 2010). The Petri Net conferences serve as annual meeting places to discuss the progress in the field of Petri nets and related models of concurrency. They provide a forum for researchers to present and discuss both applications and theoretical developments in this area. Novel tools and substantial enhancements to existing tools can also be presented. The satellite program of the conference comprised five workshops and three tutorials. This year, the conference was co-located with the 10th International Conference on Application of Concurrency to System Design (ACSD 2010). The two conferences shared five invited speakers. Detailed information about PETRI NETS 2010 can be found at <http://petrinets2010.di.uminho.pt>. The PETRI NETS 2010 conference was organized by the Universidade do Minho and the Instituto Polytécnico de Beja. It took place in Braga, Portugal during June 21-25, 2010. We would like to express our deepest thanks to the Organizing Committee, chaired by João M. Fernandes, for the time and effort invested in the conference and for all the help with local organization. We are also grateful for the financial support by Centro de Ciências e Tecnologias de Computação.

This year the number of submitted papers amounted to 50, which included 43 full papers and 7 tool papers. The authors of the papers come from 20 different countries. We thank all the authors who submitted their papers. Each paper was reviewed by at least four referees. The Program Committee (PC) meeting took place in Warszawa, Poland. At the PC meeting, 14 PC members were present who selected 16 papers, classified as: theory papers (10 accepted), application papers (2 accepted), and tool papers (4 accepted). After the conference, some authors were invited to publish an extended version of their contribution in the journal *Fundamenta Informaticae*. We wish to thank the PC members and other reviewers for their careful and timely evaluation of the submissions before the meeting. Special thanks are due to Frank Holzwarth (Springer) and Stephan Windmüller (University of Dortmund) for their friendly attitude and technical support with the Online Conference Service. Finally, we wish to express our gratitude to the invited speakers David Harel, who gave the first Distinguished Carl Adam Petri lecture, and Kim Gulstrand Larsen, Gabriel Juhás, Maciej Koutny, and Lars M. Kristensen for their contribution. As usual, the Springer LNCS team provided high-quality support in the preparation of this volume.

April 2010

Johan Lilius  
Wojciech Penczek

# Organization

## Steering Committee

Wil van der Aalst, The Netherlands	Chuang Lin, China
Jonathan Billington, Australia	Wojciech Penczek, Poland
Gianfranco Ciardo, USA	Carl Adam Petri, Germany (honorary member)
Jörg Desel, Germany	Lucia Pomello, Italy
Susanna Donatelli, Italy	Wolfgang Reisig, Germany
Serge Haddad, France	Grzegorz Rozenberg, The Netherlands
Kunihiko Hiraishi, Japan	Manuel Silva, Spain
Kurt Jensen, Denmark (Chair)	Antti Valmari, Finland
H.C.M. Kleijn, The Netherlands	Alex Yakovlev, UK
Maciej Koutny, UK	

## Program Committee

M. Bednarczyk, Poland	M. Koutny, UK
M. Bonsangue, The Netherlands	L.M. Kristensen, Norway
R. Bruni, Italy	C. Lakos, Australia
D. Buchs, Switzerland	J. Lilius, Finland (Co-chair)
P. Chrzastowski-Wachtel, Poland	C. Lin, China
G. Ciardo, USA	T. Miyamoto, Japan
J. Desel, Germany	D. Moldt, Germany
R. Devillers, Belgium	M. Mukund, India
J.M. Fernandes, Portugal	W. Penczek, Poland (Co-chair)
G. Franceschinis, Italy	L. Petrucci, France
Q.W. Ge, Japan	L. Pomello, Italy
S. Haddad, France	O-H. Roux, France
M. Heiner, Germany	N. Sidorova, The Netherlands
R. Janicki, Canada	V. Valero, Spain
G. Juhas, Slovak Republic	A. Valmari, Finland
J. Júlvez, Spain	K. Wolf, Germany
E. Kindler, Denmark	A. Yakovlev, UK

## Organizing Committee Chair

João M. Fernandes	Universidade do Minho, Portugal
-------------------	---------------------------------

## Tools Exhibition Chair

Ricardo J. Machado	Universidade do Minho, Portugal
--------------------	---------------------------------

## Publicity Chair

João Paulo Barros

Instituto Politecnico de Beja, Portugal

## Website Manager

Rui Pais

Instituto Politecnico de Beja, Portugal

## Referees

Paolo Baldan

Kamel Barkaoui

João Paulo Barros

Marco Beccuti

Béatrice Bérard

Robin Bergenthum

Giovanni Bernardi

Luca Bernardinello

Jonathan Billington

Filippo Bonchi

Anne Bouillard

Olivier Boutin

Carmen Bratosin

Lawrence Cabac

Javier Campos

Andrea Corradini

Anikó Costa

Philippe Darondeau

David De-Frutos

Zuohua Ding

Boudewijn van Dongen

Michael Duvigneau

Stefan Dziembowski

Johan Ersfolk

Jude Ezeobiejesi

Dirk Fahland

Carlo Ferigato

Fabio Gadducci

Guy Gallasch

Gilles Geeraerts

Luís Gomes

Susumu Hashizume

Kees van Hee

Marcin Hewelt

Lom Hillah

Kunihiko Hiraishi

Steve Hostettler

Kathrin Kaschner

Kais Klai

Jetty Kleijn

Michael

Koehler-Bussmeier

Fabrice Kordon

Christian Krause

Ivan Lanese

Slawomir Lasota

Didier Lime

Alban Linard

Fei Liu

Lin Liu

Alberto Lluch Lafuente

Niels Lohmann

Juan-Pablo Lopez-Grao

Levi Lucio

Ricardo Machado

Hermenegilda Macià

Cristian Mahulea

Elisabetta Mangioni

Alexis Marechal

Marco Mascheroni

Thierry Massart

Shiro Masuda

Robin Bergenthum

Hernán Melgratti

José Merseguer

Andrey Mokhov

Patrice Moreaux

Morikazu Nakamura

Tatsushi Nishi

Olivia Oanea

Atsushi Ohta

Sérgio Oliveira

Vincent Peng

G. Michele Pinna

Jean-François

Pradat-Peyre

José Quenum

Ashur Rafiev

Óscar Ribeiro

Matteo Risoldi

Christian Rohr

Fernando Rosa

Martin Schwarick

Christian Stahl

Shigemasa Takai

Satoshi Taoka

Louis-Marie Traonouez

Nikola Trcka

Jean-Baptiste Voron

Thomas Wagner

Daniela Weinberg

Jan Martijn van der Werf

Matthias

Wester-Ebbinghaus

Michael Westergaard

Harro Wimmel

Fei Xia

Jin Xiaoqing

Shingo Yamaguchi

Samir Youcef

Yang Zhao

# Table of Contents

## Invited Papers

Instance Deadlock: A Mystery behind Frozen Programs . . . . .	1
<i>Gabriel Juhás, Igor Kazlov, and Ana Juhásová</i>	
Some Thoughts on Behavioral Programming – Distinguished Carl Adam Petri Lecture (Abstract) . . . . .	18
<i>David Harel</i>	
Petri Nets with Localities and Testing . . . . .	19
<i>Jetty Kleijn and Maciej Koutny</i>	
A Perspective on Explicit State Space Exploration of Coloured Petri Nets: Past, Present, and Future . . . . .	39
<i>Lars M. Kristensen</i>	

## Full Papers

Can Stubborn Sets Be Optimal? . . . . .	43
<i>Antti Valmari and Henri Hansen</i>	
Efficient Computation of Causal Behavioural Profiles Using Structural Decomposition . . . . .	63
<i>Matthias Weidlich, Artem Polyvyanyy, Jan Mendling, and Mathias Weske</i>	
Canonical Transition Set Semantics for Petri Nets . . . . .	84
<i>Yunhe Wang and Li Jiao</i>	
A Characterization of Combined Traces Using Labeled Stratified Order Structures . . . . .	104
<i>Dai Tri Man Lê</i>	
Integrated Process Planning and Supply Chain Configuration for Commodity Assemblies Using Petri Nets . . . . .	125
<i>Oleg Gusikhin and Erica Klampfl</i>	
The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification . . . . .	145
<i>Christine Choppy, Anna Dedova, Sami Evangelista, Silien Hong, Kais Klai, and Laure Petrucci</i>	
Factorization Properties of Symbolic Unfoldings of Colored Petri Nets . . . . .	165
<i>Thomas Chatain and Eric Fabre</i>	



Forward Analysis for Petri Nets with Name Creation . . . . .	185
<i>Fernando Rosa-Velardo and David de Frutos-Escrig</i>	
Learning Workflow Petri Nets . . . . .	206
<i>Javier Esparza, Martin Leucker, and Maximilian Schlund</i>	
Process Mining from a Basis of State Regions . . . . .	226
<i>Marc Solé and Josep Carmona</i>	
Separability in Persistent Petri Nets . . . . .	246
<i>Eike Best and Philippe Darondeau</i>	
New Algorithms for Deciding the Siphon-Trap Property . . . . .	267
<i>Olivia Oanea, Harro Wimmel, and Karsten Wolf</i>	
 <b>Tool Papers</b>	
ALPiNA: A Symbolic Model Checker . . . . .	287
<i>Didier Buchs, Steve Hostettler, Alexis Marechal, and Matteo Risoldi</i>	
Wendy: A Tool to Synthesize Partners for Services . . . . .	297
<i>Niels Lohmann and Daniela Weinberg</i>	
GreatSPN Enhanced with Decision Diagram Data Structures . . . . .	308
<i>Junaid Babar, Marco Beccuti, Susanna Donatelli, and Andrew Miner</i>	
PNML Framework: An Extendable Reference Implementation of the Petri Net Markup Language . . . . .	318
<i>L.M. Hillah, F. Kordon, L. Petrucci, and N. Trèves</i>	
<b>Author Index</b> . . . . .	329

# Instance Deadlock: A Mystery behind Frozen Programs

Gabriel Juhás, Igor Kazlov, and Ana Juhásová

Faculty of Electrical Engineering and Information Technology  
Slovak University of Technology, Bratislava, Slovakia  
gabriel.juhas@stuba.sk

**Abstract.** In the paper we discuss the event-driven reactive programs and systems, which does not deadlock for one instance, but because of shared resources, can deadlock for several instances. We focus on event-driven programs, where instances have a correct finish, and resources can be used by single instances, but can neither be destroyed nor created by instances. Typical examples include workflow processes, where each case creates an instance of the process and instances share resources used to execute single activities. Formally, we model such event-driven programs and systems by workflow nets, enriched by so called static places, introduced in [3] as resource constrained workflow nets (rcwf-nets). We investigate, whether an rcwf-net, which is sound for a single instance is sound for multiple instances (dynamically sound) or whether it contains an instance deadlock for a number of instances. We show that the detection of instance deadlock and the dynamic soundness of rcwf-nets is decidable by transforming the problem to bounded place/transition Petri nets.

## 1 Introduction

When you enter the key words "frozen computer" into the Google search engine, the first link you get is a web page eHow dealing with how to fix computer freezes. As one of the suggestions it writes: "Close applications, run only what you need to. Computer freezes happen when too many software programs run at the same time." In fact, not only computer programs, but event-driven reactive systems in general, such as communication networks and workflow processes often hang/crash because of a deadlock, even if each running instance of a program itself is deadlock-free. The reason is that the programs/processes/threads use shared resources (memory, ports, processors, buses etc. in case of computers and networks, people or machines in case of bussines processes or flexible manufacturing systems). Thus, in this paper we focus on a special case of a deadlock. We consider event-driven reactive systems, which does not deadlock for a single instance, but because of shared resources, can deadlock for multiple instances.

In terminology of object-oriented programs, we restrict ourselves to event-driven programs implemented by a class consisting of a set of attributes (variables) and a set of methods that change the attributes, and an event listener. Shared resources are implemented via static variables. Using the event listener, an instance of the program is reacting to an event occurrence by calling some of the enabled methods. Thus, a method can be called by the event listener just if attributes have defined values, otherwise it is disabled. Furthermore, we focus on event-driven systems, where instances have a correct finish, and resources can be used by single instances, but can neither be destroyed

nor created by instances. During the run time of the system, instances are created and after correct finish they are closed, thus none or several instances can be running. We consider that a system itself is correct for one instance, i.e. any single instance can be finished without deadlocking before correct finish. We investigate the question, whether the system deadlocks for multiple instances, i.e. we ask whether there is a number of instances, for which each running instance deadlocks without finishing correctly, i.e. the values of static variables and variables of each running instance cause that all methods of all running instances are disabled. Typical examples include workflow processes, where each processed case creates a new instance of the process and instances share resources used to execute single activities. Workflow nets (wf-nets) are a prominent formalism to model and analyze workflow processes [1]. Their main power in comparison with other modelling techniques for workflow processes is the possibility to find errors in processes in a formal way [4]. Classical wf-nets do not consider soundness analysis and detection of a deadlock for multiple instances (cases) and constrained number of resources. In practice, the question of soundness considering multiple instances and constrained number of resources remains crucial. To formalize the problem we follow the concept of static and non-static places introduced in [3] resulting in resource constrained workflow nets.

When understood as a simple program, an rcwf-net can be understood as a class definition, with static places corresponding to the static variables shared by all instances (i.e. static places are holders for different types of shared resources) and non-static places as well as transitions corresponding to the non-static variables and methods of instances, which are no more shared by single instances. Thus, each case creates a new instance with its own copy of non-static places, transitions and respective arcs. Following [3], we consider a situation, where a marking of a static place is bounded by the initial value and it remains unchanged after processing a single instance. We provide a definition which is fully based on place/transition nets (low-level Petri nets) [5], in comparison to [3], where coloured nets (high level Petri nets) [6] with token id-s are used.

In the paper we investigate, whether an rcwf-net, which is sound for a single instance is sound for multiple instances (dynamically sound) or whether it contains an instance deadlock for a number of instances. We show that the detection of instance deadlock and the dynamic soundness of rcwf-nets is decidable by transforming the problem to bounded place/transition Petri nets.

## 2 Place/Transition Nets and Workflow Nets

In this section we introduce basic notions for place/transition nets and workflow nets following [5] and [1]. As usual, we use  $\mathbb{N}$  to denote the nonnegative integers and  $\mathbb{N}^+$  to denote positive integers and by  $a \div b$  for given integers  $a, b$  we denote the quotient of integer division. Given two sets  $A$  and  $B$  we use  $B^A$  to denote all functions from  $A$  to  $B$ . Given a set  $A$ , a number  $n \in \mathbb{N}$ , and a (nonempty) sequence  $\sigma : \{1, \dots, n\} \rightarrow A$  of elements from  $A$ , we define  $|\sigma| = n$  to denote the number of elements of  $\sigma$ .

A sequence  $f : \{1, \dots, k\} \rightarrow \{1, \dots, n\}$  such that  $k, n \in \mathbb{N}^+$ ,  $1 \leq k \leq n$  and  $i < j \Rightarrow f(i) < f(j)$  for every  $i, j \in \{1, \dots, k\}$  is called a subsequence of a sequence with  $n$  elements. We denote  $\Psi_n$  the set of all subsequences of a sequence with  $n$  elements.

**Definition 1. (Place/transition net)**

A place/transition net (shortly a p/t net) is a quadruple  $(P, T, F, W)$ , where  $P$  is a finite set of places,  $T$  is a finite set of transitions such that  $P \cap T = \emptyset$ ,  $F \subseteq (P \times T) \cup (T \times P)$  is a flow relation and  $W : F \rightarrow \mathbb{N}$  is a weighth function.

As usual, we extend the weight function  $W$  to pairs of net elements  $(x, y)$  satisfying  $(x, y) \notin F$  by  $W(x, y) = 0$ .

Let  $(P, T, F, W)$  be a p/t net and let  $x \in P \cup T$ . Then  $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$  is called preset of  $x$  and similarly  $x \bullet = \{y \in P \cup T \mid (x, y) \in F\}$  is called postset of  $x$ . Marking of a p/t net  $(P, T, F, W)$  is a function  $m : P \rightarrow \mathbb{N}$ .

A pair  $(N, m_0)$ , where  $N = (P, T, F, W)$  is a p/t net and  $m_0 : P \rightarrow \mathbb{N}$  is its marking, is called a marked p/t net and  $m_0$  is referred as the initial marking of the marked p/t net.

Places are graphically depicted by circles, transitions by boxes, elements of the flow relation by arcs, values of the weighth function by labels of arcs and values of a marking of a place by the appropriate number of black tokens in the place. As usual, labels of the arcs corresponding to the values of the weighth function equal to 1 are omitted.

**Definition 2. (Occurrence rule, Deadlock)**

Let  $N = (P, T, F, W)$  be a p/t-net. A transition  $t \in T$  is enabled to occur in a marking  $m$  of  $N$  if  $m(p) \geq W(p, t)$  for every place  $p \in \bullet t$ , and then we say that  $m$  enables  $t$  in  $N$ , otherwise  $t$  is said to be disabled to occur in the marking  $m$  of  $N$  and we say that  $m$  disables  $t$  in  $N$ .

If a marking  $m$  disables any  $t \in T$  in  $N$  then we say that  $m$  is a deadlock of  $N$ .

If a transition  $t$  is enabled to occur in a marking  $m$ , then its occurrence leads to the new marking  $m'$  defined by  $m'(p) = m(p) - W(p, t) + W(t, p)$  for every  $p \in P$ . We write  $m \xrightarrow{t} m'$  to denote that  $t$  is enabled to occur in  $m$  and that its occurrence leads to  $m'$ .

**Definition 3. (Occurrence sequence, Reachability)**

Let  $N = (P, T, F, W)$  be a p/t-net and  $m$  be a marking of  $N$ . Then  $m$  is said to be reachable from  $m$ . A finite and nonempty sequence of transitions  $\sigma : \{0, \dots, n\} \rightarrow T$  is called an occurrence sequence enabled in  $m$  and leading to  $m'$  if there exists a sequence of markings  $\rho : \{1, \dots, n-1\} \rightarrow \mathbb{N}^P$  such that

$$m \xrightarrow{\sigma(1)} \rho(1) \xrightarrow{\sigma(2)} \dots \rho(n-1) \xrightarrow{\sigma(n)} m'.$$

Then the marking  $m'$  is said to be reachable from the marking  $m$ . We write  $m \xrightarrow{\sigma} m'$  to denote that  $\sigma$  is enabled to occur in  $m$  and that its occurrence leads to  $m'$ .

In a marked p/t-net, the markings reachable from the initial marking  $m_0$  are shortly called reachable markings. The set of all markings reachable from  $m_0$  is denoted by  $[m_0]$ . The set of all reachable markings which are deadlocks is denoted by  $[m_0]_{dead}$ .

**Definition 4. (Workflow net)**

A p/t net  $PN = (P, T, F, W)$  is called a workflow net (shortly a wf-net) iff: There exist a place  $in \in P$  with  $\bullet in = \emptyset \wedge in \bullet \neq \emptyset$  called an input place and a place  $out \in P$

with  $out^\bullet = \emptyset \wedge \bullet out \neq \emptyset$  called an output place. Moreover,  $\forall p \in P \setminus \{in, out\} : \bullet p \neq \emptyset \wedge p^\bullet \neq \emptyset$ . Let  $m_{in}$  denote the marking of the wf-net such that  $m_{in}(in) = 1$  and  $m(p) = 0$  for each  $p \in P \setminus \{in\}$  and  $m_{out}$  denote the marking of the wf-net such that  $m_{out}(out) = 1$  and  $m(p) = 0$  for each  $p \in P \setminus \{out\}$ .

In the following definition of soundness we relax the requirement that each transition has to be used at least in one branch of the process, as this requirement is not important for our further discussion.

**Definition 5. (Soundness of a wf-net):** A wf-net  $PN = (P, T, F, W)$  is sound if for each  $m$  reachable from  $m_{in}$  there holds

- $m_{out}$  is reachable from  $m$  and
- if  $m(out) = 1$  then  $m = m_{out}$ , i.e.  $m_{out}$  is the only marking reachable from  $m_{in}$  with marked output place.

A well known fact is that whenever a wf-net is sound then it has finite number of reachable markings, see e.g. [1].

**Lemma 1.** Let  $PN$  be a wf-net. If  $PN$  is sound then the number of markings reachable from  $m_{in}$  in  $PN$  is finite.

### 3 Resource Constrained Workflow Nets

In this section we introduce resource constrained workflow nets motivated by [3]. In comparison to [3] we allow more types of resources. Following [3], we require that the number of resources is bounded by the initial value and it remains unchanged after processing any single instance.

**Definition 6.** Let  $PN = (P, T, F, W)$  be a wf-net. Let  $P = S \cup D$  with  $S \cap D = \emptyset$  and  $in, out \in D$ . Set  $D$  denotes non-static places and set  $S$  denotes the static places (shared resource holders). Let  $m_0$  be a marking of  $PN$  such that  $m_0(d) = 0$  for each  $d \in D \setminus \{in\}$  and  $m_0(in) = 1$ . Then marked p/t net  $(PN, m_0)$  is called resource-constrained wf-net (shortly rcwf-net). By  $m_f$  we denote the marking of rcwf-net called final marking given by:  $m_f(d) = 0$  for each  $d \in D \setminus \{out\}$ ,  $m_f(out) = 1$  and  $m_f(s) = m_0(s)$  for each  $s \in S$ .

Static places (also called resource places in [3]) are depicted in figures by circles with shadow.

An example of an rcwf-net is given in Figure 1. It models a simple process of treating patient in an emergency department. Marking of the static place  $s1$  stands for the number of available doctors. Transition  $t1$  stands for a decision that a case is complicated and two doctors are necessary to treat the patient, therefore it produces 2 tokens to place  $d1$ . Transition  $t3$  means that a doctor goes to treat a patient who needs two doctors. Number of tokens in place  $d3$  stands for the number of doctors treating the patient. Transition  $t5$  means that two doctors finished to treat the patient and are available again. The branch with places and transitions indexed by even numbers model the treating of a patient where only one doctor is needed.

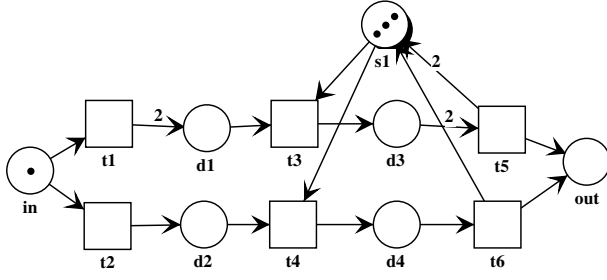


Fig. 1. an rcwf-net

**Definition 7. (Single instance soundness of an rcwf-net)**

An rcwf-net  $(PN, m_0)$  is sound for a single instance if for each  $m$  reachable from  $m_0$  there holds:

- $m_f$  is reachable from  $m$ , and
- $\forall s \in S : m(s) \leq m_0(s)$ , and
- if  $m(out) = 1$  then  $m = m_f$ .

Using lemma [1](#) we get that a necessary condition for dynamic soundness of an rcwf-net is that it has a finite number of reachable markings.

**Corollary 1.** Let  $(PN, m_0)$  be an rcwf-net. If  $(PN, m_0)$  is sound for a single instance then the number of markings reachable from  $m_0$  in  $(PN, m_0)$  is finite.

Obviously, at a given moment of time several instances (cases) of a process described by an rcwf-net can be handled. This is expressed by so called run-time nets of rcwf-nets.

**Definition 8. (Run-time nets of an rcwf-net)**

Let  $(PN = (P = (S \cup D), T, F, W), m_0)$  be an rcwf-net and  $n \in \mathbb{N}^+$ .

Let  $P^n = S \cup (D \times \{1, \dots, n\})$ ,  $T^n = T \times \{1, \dots, n\}$ ,

$F_1^n = \{((x, i), (y, i)) \in ((P^n \setminus S) \times T^n) \cup (T^n \times (P^n \setminus S)) \mid (x, y) \in F\}$ ,

$F_2^n = \{(x), (y, i) \in (S \times T^n) \mid (x, y) \in F\}$ ,

$F_3^n = \{(x, i), (y) \in (T^n \times S) \mid (x, y) \in F\}$ ,

$F^n = F_1^n \cup F_2^n \cup F_3^n$

and let  $W^n : F^n \rightarrow \mathbb{N}$  be given by:

$\forall ((x, i), (y, i)) \in F_1^n : W^n((x, i), (y, i)) = W(x, y)$

$\forall (x), (y, i) \in F_2^n : W^n(x, (y, i)) = W(x, y)$

$\forall ((x, i), (y)) \in F_3^n : W^n((x, i), y) = W(x, y)$ .

Let  $m_0^n : P^n \rightarrow \mathbb{N}$  denote a marking satisfying:

$m_0^n(s) = m_0(s)$  for each  $s \in S$  and  $m_0^n(d, i) = m_0(d)$  for each  $(d, i) \in D \times \{1, \dots, n\}$ .

Then the marked p/t net  $(PN^n = (P^n, T^n, F^n, W^n), m_0^n)$  is called the run-time net of rcwf-net  $(PN, m_0)$  for  $n$  instances.

By  $m_f^n$  we denote the marking of run-time net  $(PN^n, m_0^n)$  of rcwf-net  $(PN, m_0)$  called final marking given by:  $m_f^n(d) = 0$  for each  $d \in P^n \setminus (S \cup (\{out\} \times \{1, \dots, n\}))$ ,  $m_f^n(out, i) = 1$  for each  $i \in \{1, \dots, n\}$  and  $m_f^n(s) = m_0(s)$  for each  $s \in S$ .

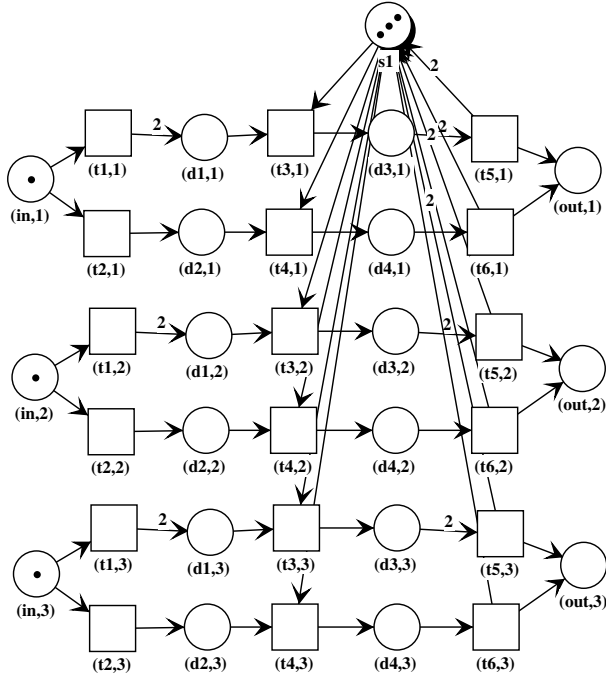


Fig. 2. Run-time net for 3 instances

Dynamic soundness expresses the property saying that for any number of instances the process modelled by respective run-time net will terminate properly, i.e. the process is sound for a single instance and has no instance deadlock.

**Definition 9. (Instance deadlock of rcwf-nets)**

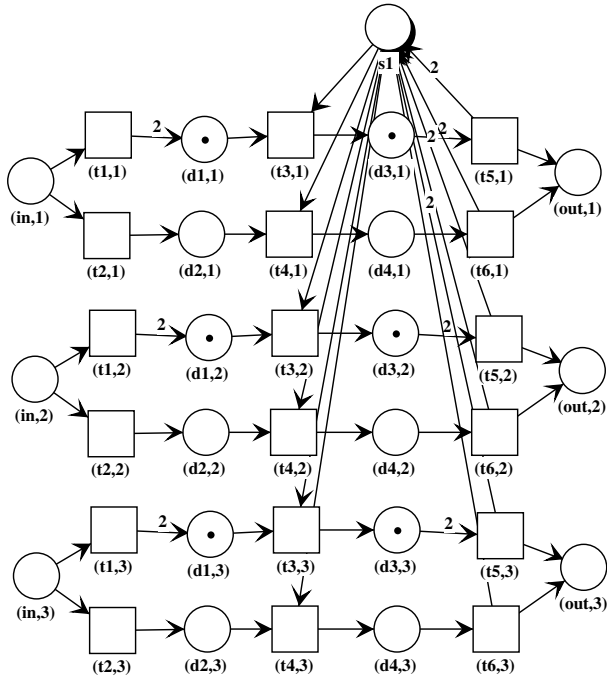
Let  $n \in \mathbb{N}^+$  and  $(PN, m_0)$  be an rcwf-net sound for a single instance. Let  $(PN^n, m_0^n)$  be the run-time net of rcwf-net  $(PN, m_0)$  for  $n$  instances. A marking  $m^n : P^n \rightarrow \mathbb{N}$  is called an instance deadlock of rcwf-net  $(PN, m_0)$  for  $n$  instances if  $m^n \in [m_0^n]_{dead} \setminus \{m_f^n\}$ . If  $[m_0^n]_{dead} = \{m_f^n\}$  then we say that rcwf-net  $(PN, m_0)$  has no instance deadlock (is instance deadlock free) for  $n$  instances.

An rcwf-net  $(PN, m_0)$  has no instance deadlock (is instance deadlock free) if for each  $n \in \mathbb{N}$  there holds:  $(PN, m_0)$  has no instance deadlock for  $n$  instances.

**Definition 10. (Dynamic soundness of rcwf-nets)**

Let  $(PN, m_0)$  be an rcwf-net sound for a single instance.  $(PN, m_0)$  is sound for  $n$  instances if it has no instance deadlock for  $n$  instances.  $(PN, m_0)$  is dynamically sound if it has no instance deadlock.

As it is illustrated in Figure 3, the rcwf-net from Figure 1, which is sound for 1 instance is not dynamically sound as it has the instance deadlock for 3 instances depicted in Figure 3. The question is, whether the dynamic soundness is decidable in general.

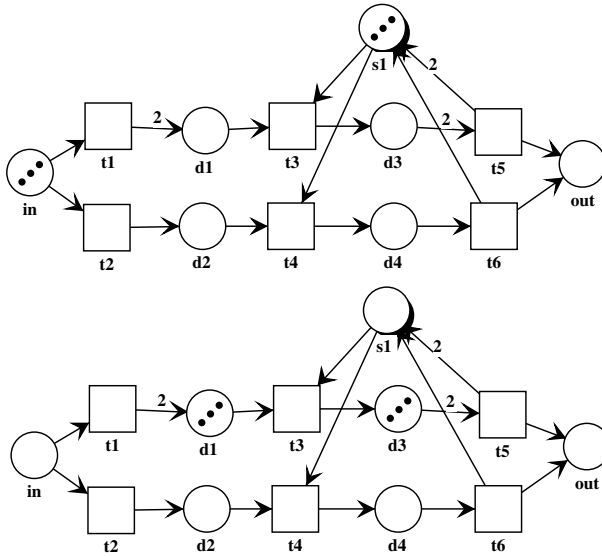


**Fig. 3.** Deadlock of the run-time net for 3 instances after firing  $(t1, 1)$ ,  $(t1, 2)$ ,  $(t1, 3)$  and  $(t3, 1)$ ,  $(t3, 2)$ ,  $(t3, 3)$

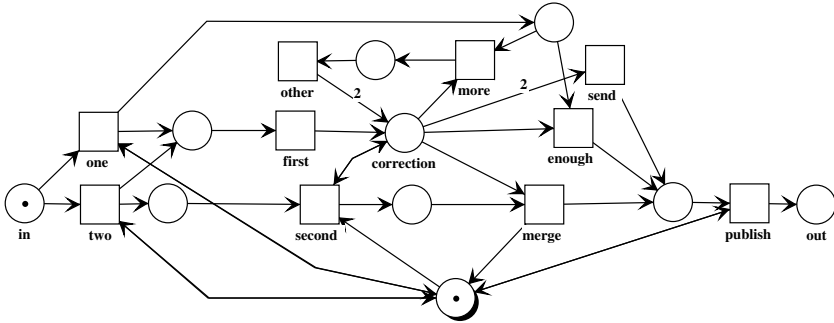
The problem is caused by the fact that one has to check infinitely many run-time nets. Obviously, they all can be modelled by a single coloured Petri net [6] with infinitely many token colours. However, the problem of reachability in coloured nets with infinitely many tokens is undecidable. One naive approach is to model multiple instances by multiple tokens in the initial place of the rcwf-net. As the example in Figure 4 illustrates, this can cause that tokens of different instances will be mixed and therefore a transition can occur which cannot occur for any single instance. Thus, an existing instance deadlock for 3 instances illustrated in Figure 3 would not be detected using 3 tokens in the place  $p_{in}$  (Figure 4), because after firing tree times  $t1$  and  $t3$  we get the 3 tokens representing different instances in place  $d3$  and the transition  $t5$  can occur, which cannot occur for any single instance.

As the example in Figure 5 illustrates, a nonexisting instance deadlock could be detected. Namely, the model of an (artificial) workflow process of correcting and publishing a manuscript on a web page is described. The web page has one editor. We suppose unlimited number of correctors. For short manuscripts, only one corrector is necessary, corresponding to an occurrence of transition *one* representing the decision of the editor. A corrector corrects the manuscript (transition *first*), and if there were not many errors, transition *enough* occurs and finally the editor publishes the manuscript (transition *publish* occurs). If there were many errors, the manuscript is corrected once again (transitions *more* and *other*). After that the manuscript is sent to the editor who publishes





**Fig. 4.** Deadlock of the system for 3 instances after firing  $t1$  three times and  $t3$  three times will not be detected



**Fig. 5.** An instance deadlock free net

it. If a manuscript is longer, the editor decides to require two corrections (transition *two*), first performed by a corrector (transition *first*) and after it is finished the second correction is performed by the editor himself (transition *second*). After the corrections are finished, the editor immediately merges them to one document (transition *merge*) and publishes it (transition *publish*).

Observe that the process is dynamically sound, i.e. it has no instance deadlock. But if one consider for the instance deadlock detection the same net with two tokens in the input place *in* (Figure 6), then for example the firing sequence of transitions *one*, *two*, *first*, *second*, *first* enables transition *send* by mixing tokens from two instances in place *correction*, which causes the deadlock depicted in Figure 7. Such deadlock will never happen in the system.

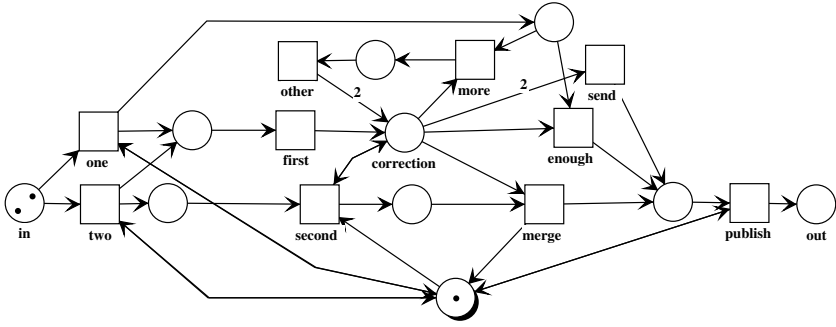


Fig. 6. Modelling two instances with low level tokens

Thus, modelling multiple instances with several low level tokens in the original net does not work in general.

The paper [3] gives an algorithm for the decision of dynamic soundness for a restricted class of rcfw-nets with just one static place. The paper [3] also mentions that an rcfw-net with one static place can be transformed to a state machine via its reachability graph and claims that the original net is sound iff the transformed is. Unfortunately, [3] does not formalize the transformation. Moreover, the mentioned result and the algorithm for checking soundness is provided for state machines with one static place only and the general case with more than one static place is left in [3] to be a problem for future work. Here we formalize the transformation mentioned in [3] and show that dynamic soundness is decidable in general. Actually, in the transformed net each reachable marking is replaced by a place, each transition of the rcfw-net enabled in a marking of the reachability graph is replaced by a copy transition in the transformed net and the static places are connected to a copy transition whenever they are connected to the original transition of the rcfw-net. In comparison to [3], we add to the transformed net a constructor transition *new*, a marked place *source*, which serves as a source for the constructor transition *new*, and a transition *stop*, which removes a token from *source* and disables *new*.

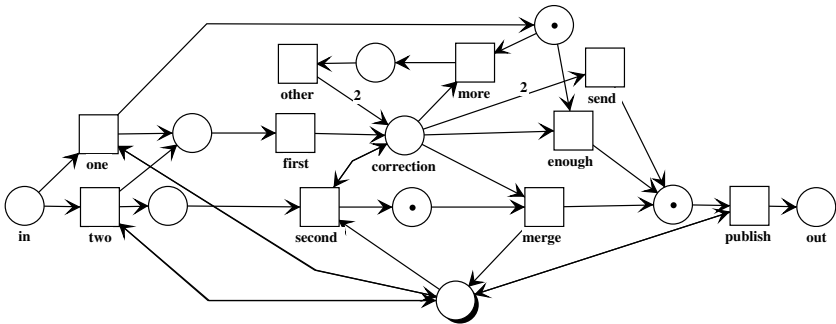


Fig. 7. Finding non existing deadlock

**Definition 11. (Transformed rcwf-net)**

Let  $(PN = (P = S \cup D, T, F, W), m_0)$  be an rcwf-net with a finite number of markings reachable from  $m_0$ . Let  $source, new, stop \notin S \cup [m_0] \cup ([m_0] \times T \times [m_0])$ .

Define  $P^{tr} = S \cup [m_0] \cup \{source\}$ ,

$T^{tr} = \{(m_1, t, m_2) \in [m_0] \times T \times [m_0] \mid m_1 \xrightarrow{t} m_2\} \cup \{new, stop\}$ ,

$F^{tr} = \{(m, (m_1, t, m_2)) \in ([m_0] \times T^{tr}) \mid m = m_1\} \cup$

$\{(m_1, t, m_2), m) \in (T^{tr} \times [m_0]) \mid m = m_2\} \cup$

$\{(s, (m_1, t, m_2)) \in (S \times T^{tr}) \mid (s, t) \in F\} \cup$

$\{(m_1, t, m_2), s) \in (T^{tr} \times S) \mid (t, s) \in F\} \cup$

$\{(source, stop), (source, new), (new, source), (new, m_0)\}$ .

Let  $W^{tr} : F^{tr} \rightarrow \mathbb{N}$  be given as follows:

$W^{tr}(x, (m_1, t, m_2)) = W(x, t)$  and  $W^{tr}((m_1, t, m_2), x) = W(t, x)$  whenever  $x \in S$ , otherwise  $W^{tr}(f) = 1$ .

Let  $\mu_0 : P^{tr} \rightarrow \mathbb{N}$  be defined by:  $\mu_0(s) = m_0(s)$  for each  $s \in S$ ,  $\mu_0(source) = 1$  and  $\mu_0(m) = 0$  for each  $m \in [m_0]$ .

Then the marked p/t net  $(PN^{tr} = (P^{tr}, T^{tr}, F^{tr}, W^{tr}), \mu_0)$  is called the transformed net of the rcwf-net  $(PN, m_0)$ .

By  $\Phi^{tr}$  we denote the set of final markings of  $(PN^{tr}, \mu_0)$  given by

$\Phi^{tr} = \{\mu \in [\mu_0] \mid (\forall s \in S : \mu(s) = m_0(s)) \wedge (\forall x \in P^{tr} \setminus S : \mu(x) \neq 0 \Rightarrow x = m_f)\}$ .

The example of the transformed net of the rcwf-net from Figure 1 is in Figure 8.

Markings of places from  $[m_0]$  represent how many instances are in the given state.

For the rest of the paper let us consider that an rcwf-net  $(PN, m_0)$  sound for a single instance and its transformed net  $(PN^{tr}, \mu_0)$  are given using the notation from Definition 11. To avoid confusion, we will use  $\mu$  for markings of the transformed net and  $\tau$  to denote transitions of the transformed net. From the definition of transformed nets we get the following result.

**Lemma 2**

$(PN, m_0)$  is dynamically sound (i.e. it has no instance deadlock) iff  $[m_0]_{dead} = \Phi^{tr}$ .

In fact, the transformed net is unbounded, but its structure can be used for a further simplification of the problem. Namely, one can ask whether there exists for a given

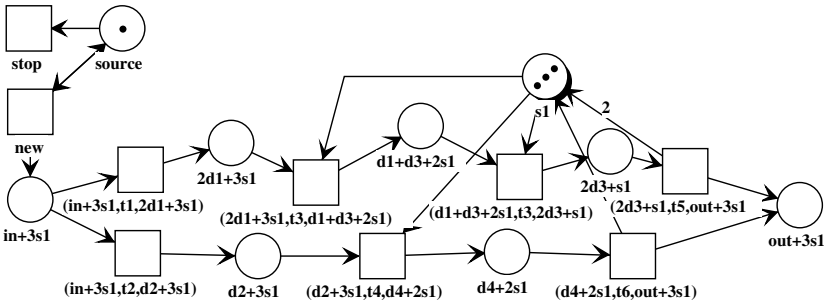


Fig. 8. The transformed rcwf-net

rcwf-net a number of instances  $n$  such that the net is dynamically sound if and only if it is sound for  $n$  instances. We show that the answer is positive and show one simple and one more complicated way to determine the sufficient number of instances to check the dynamic soundness. In other words, we show that the lemma [2](#) holds also if we replace the transformed net by a bounded transformed net. It means that the detection of instance deadlocks and the dynamic soundness of an rcwf-net can be investigated using the set of reachable markings of a bounded place/transition Petri net. Further, we provide a simple algorithm to determine dynamic soundness without determining the number of the instances sufficient to check it.

Because the sum of tokens in places from  $[m_0]$  of the transformed net is increased by one whenever the constructor transition  $new$  occurs, but will never decrease, we get that the reachability of a marking with total  $n$  tokens distributed in places from  $[m_0]$  can be determined by firing only sequences with  $n$  occurrences of the constructor  $new$ .

**Lemma 3.** *Let  $\mu \in [\mu_0]$ ,  $\sigma : \{1, \dots, n\} \rightarrow T^{tr}$  and  $\mu_0 \xrightarrow{\sigma} \mu$ . Then the number  $|\{i \in \{1, \dots, n\} \mid \sigma(i) = new\}|$  of occurrences of  $new$  in  $\sigma$  equals the sum  $\sum_{m \in [m_0]} \mu(m)$  of tokens in places from  $[m_0]$ .*

Another consequence of the structure is that each occurrence sequence  $\sigma$  of the transformed net is given by an interleaving of a set of occurrence sequences of the original rcwf-net, where the number of tokens in places from  $[m_0]$  gives the number of subsequences of  $\sigma$ . Obviously, not each interleaving of a set of occurrence sequences of the original net must be enabled in the transformed net, because the static places can disable some transitions.

**Lemma 4.** *Let  $\mu \in [\mu_0]$ ,  $\sigma : \{1, \dots, n\} \rightarrow T^{tr}$  and  $\mu_0 \xrightarrow{\sigma} \mu$ . Then there exists a sequence  $f : \{1, \dots, \sum_{m \in [m_0]} \mu(m)\} \rightarrow \Psi_{|\sigma|}$  of subsequences of  $\sigma$  such that*

- $\sum_{i \in \{1, \dots, |f|\}} |f(i)| = |\sigma|$ , i.e. the sum of the transitions in all subsequences equals the number of transitions of  $\sigma$ , and
- for each  $i, j \in \{1, \dots, |f|\}$  there holds:  
 $i \neq j \Rightarrow (\forall k \in \{1, \dots, |f(i)|\}, \forall l \in \{1, \dots, |f(j)|\} : f(i)(k) \neq f(j)(l))$ , i.e. different subsequences do not contain the same elements of  $\sigma$ , and
- for each  $i \in \{1, \dots, |f|\}$  there holds:  $\forall a \in \{2, \dots, |f(i)|\} : \pi_3(\sigma(f(i)(a-1))) = \pi_1(\sigma(f(i)(a)))$  where  $\pi_1, \pi_3 : [m_0] \times T \times [m_0] \rightarrow [m_0]$  are projections given by  $\pi_1(m, t, m') = m$  and  $\pi_3(m, t, m') = m'$  for each  $(m, t, m') \in [m_0] \times T \times [m_0]$ , i.e. the subsequences determine the occurrence sequences of the original net and finally,
- $\forall m \in [m_0] : \mu(m) = |\{i \in \{1, \dots, |f|\} \mid \pi_3(\sigma(f(i)(|f(i)|))) = m\}|$ , i.e. the number of subsequences finishing in  $m$  equals the number of tokens in  $m$ .

Using the fact that the static places are bounded by their initial value, i.e. for each  $s \in S$  and each  $m \in [m_0]$  there holds:  $m_0(s) - m(s) \geq 0$ , we get the following statement.

**Lemma 5.** *For each  $\mu \in [\mu_0]$  and each  $s \in S$  there holds:*

$$0 \leq \mu(s) = \mu_0(s) - \sum_{m \in [m_0]} \mu(m) \cdot (m_0(s) - m(s)) \leq \mu_0(s) = m_0(s).$$

The previous statement says, that the static places are bounded by the initial value also in the transformed net. Moreover, it says that the markings of static places are fully determined by the markings of non-static places of the transformed net which means that there cannot be two different reachable markings of the transformed net which differs only in static places.

However, because of the previous lemmas, whenever a marking of the transformed net is reachable, than any marking of the transformed net with less or equal tokens in non-static places and with static places marked as given by lemma 5 is reachable.

**Theorem 1.** *Let  $\mu \in [\mu_0]$  and let  $\mu' : P^{tr} \rightarrow \mathbb{N}$  be such that*

- $\forall m \in [m_0] \cup \{\text{source}\} : \mu'(m) \leq \mu(m)$  and
- $\forall s \in S : \mu(s) = \mu_0(s) - \sum_{m \in [m_0]} \mu(m) \cdot (m_0(s) - m(s))$ .

*Then  $\mu' \in [\mu_0]$ .*

Our detection of the instance deadlocks will be based on so called dangerous markings of the original rcwf-net. A dangerous marking is any marking  $m$  reachable from  $m_0$  satisfying that there exists a static place  $s$  such that  $m(s)$  is less than  $m_0(s)$  and whenever a transition enabled in  $m$  consumes a token from  $m$  then it also consumes something from a static place.

**Definition 12. (Dangerous marking, Basic deadlock)**

*Let  $m \in [m_0]$  such that for each  $\tau \in m^\bullet$  there holds  $\bullet\tau \cap S \neq \emptyset$  and there exists  $s \in S$  such that  $m_0(s) - m(s) > 0$ . Then  $m$  is called dangerous.*

*The set of all dangerous markings of  $(PN, m_0)$  will be denoted by  $[m_0]_{dg}$ .*

*Whenever  $[m_0]_{dg} \neq \emptyset$ , we define a simple bound function  $b : [m_0]_{dg} \rightarrow \mathbb{N}$  by*

*$b(m) = \min_{s \in \{s \in S \mid m_0(s) - m(s) > 0\}} (m_0(s) \div (m_0(s) - m(s)))$  for each  $m \in [m_0]_{dg}$ .*

*We define  $[\mu_0]_{dg} = \{\mu \in [\mu_0] \mid \forall m \in (([m_0] \cup \{\text{source}\}) \setminus [m_0]_{dg}) : \mu(m) = 0\}$  to be the set of all markings of  $(PN^{tr}, \mu_0)$  that do not mark any place of  $PN^{tr}$  except dangerous markings of  $PN$  and static places.*

*Finally, we define  $[\mu_0]_{bid} = ([\mu_0]_{dg} \cap [\mu_0]_{dead}) \setminus \Phi^{tr}$  to be the set of basic deadlocks of  $(PN^{tr}, \mu_0)$  determining the instance deadlocks of  $(PN, m_0)$ .*

Obviously, because of the lemma 5, we get that the dangerous markings are bounded when used as places in the transformed net.

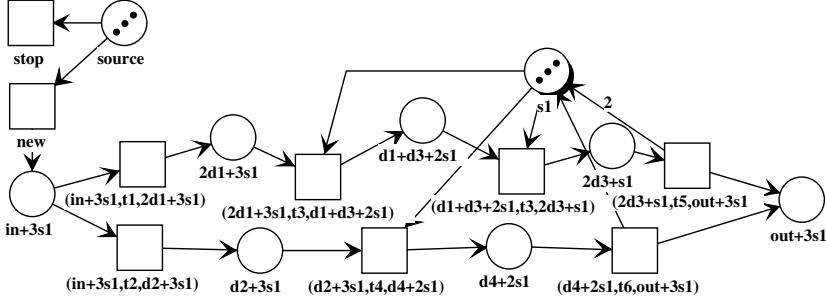
**Lemma 6**

*Let  $m$  is a dangerous marking. Then for each  $\mu \in [\mu_0]$  there holds  $\mu(m) \leq b(m)$ .*

We have the following property of the instance deadlocks, i.e. deadlocks of the transformed net different from a final marking: Given a deadlock  $\mu$  of  $PN^{tr}$  reachable from  $\mu_0$  and different from any marking from  $\Phi^{tr}$ , there exists a dangerous marking  $m$  such that  $m$  is marked in  $\mu$  and each  $\tau$  consuming from  $m$  needs to consume from a static place  $s$  but there are not enough tokens in  $s$  to enable  $\tau$ .

More precisely, we get the following lemma.

**Lemma 7.** *Let  $\mu \in [\mu_0] \setminus \Phi^{tr}$  be a deadlock of  $PN^{tr}$ . Then there exists a dangerous marking  $m$  such that  $\mu(m) > 0$  and for each  $\tau \in m^\bullet$  there exists  $s \in S$  such that  $\mu(s) < W^{tr}(s, \tau)$ .*



**Fig. 9.** The 3-bounded transformed net  $(PN_3^{tr}, \mu_0^3)$  with the only dangerous marking  $d1 + d3 + 2s1$  of  $(PN, m_0)$

Thus, dangerous markings are necessary for the existence of an instance deadlock.

**Corollary 2.** *If  $[m_0]_{dg} = \emptyset$  then  $(PN, m_0)$  has no instance deadlock.*

In fact, dangerous markings are enough to detect the existence of an instance deadlock.

**Lemma 8.** *Let  $\mu \in [\mu_0]_{dead} \setminus \Phi^{tr}$ . Let  $\mu_{dg}$  be given as follows:*

- $\forall m \in [m_0]_{dg} : \mu_{dg}(m) = \mu(m)$  and
- $\forall m \in (([m_0] \setminus [m_0]_{dg}) \cup \{source\}) : \mu_{dg}(m) = 0$  and
- $\forall s \in S : \mu_{dg}(s) = \mu_0(s) - \sum_{m \in [m_0]_{dg}} \mu(m) \cdot (m_0(s) - m(s))$ .

Then  $\mu_{dg} \in [\mu_0]_{bid}$ .

This gives the following result.

### Theorem 2

$(PN, m_0)$  is dynamically sound (i.e. it has no instance deadlock) iff  $[\mu_0]_{bid} = \emptyset$ .

Joining the previous results together, we get that it is enough to investigate at most so many instances (each instance represented by an occurrence of the constructor transition *new* in the transformed net) as many tokens can be put in dangerous markings in the transformed net. In order to simulate behaviour for a bounded number of instances, we will slightly modify the transformed net by removing the arc from *new* to *source* and increasing the number of tokens in *source* in the initial marking.

**Definition 13. (Bounded transformed net):** *Given  $n \in \mathbb{N}$ ,  $n$ -bounded transformed net is the marked p/t net  $(PN_n^{tr} = (P^{tr}, T^{tr}, F_n^{tr} = F^{tr} \setminus \{(new, source)\}), W_n^{tr} = W^{tr}|_{F_n^{tr}}, \mu_0^n)$  with  $\mu_0^n(source) = n$  and  $\mu_0^n(x) = \mu_0(x)$  for each  $x \in P^{tr} \setminus \{source\}$ . We define  $[\mu_0^n]_{dg} = \{\mu \in [\mu_0^n] \mid \forall m \in (([m_0] \cup \{source\}) \setminus [m_0]_{dg}) : \mu(m) = 0\}$ . Finally, let  $\Phi_n^{tr} = \Phi^{tr} \cap [\mu_0^n]$  and  $[\mu_0^n]_{bid} = ([\mu_0^n]_{dg} \cap [\mu_0^n]_{dead}) \setminus \Phi_n^{tr}$ .*

By the simple bound function we already determined the maximal number of tokens for any single dangerous marking. Obviously, a simple sum of those bounds gives a

sufficient number of instances for detection of an instance deadlock. Denoting by  $n$  this sum of bounds of dangerous markings, we get that an rcwf-net is sound iff its  $n$ -bounded transformed net has no basic deadlocks, i.e. deadlocks with marked only dangerous markings and static places.

**Theorem 3.** *Let  $[m_0]_{dg} \neq \emptyset$  and  $n = \sum_{m \in [m_0]_{dg}} b(m)$ . Then  $[\mu_0]_{bid} = [\mu_0^n]_{bid}$  and  $(PN, m_0)$  is dynamically sound (i.e. has no instance deadlock) iff  $[\mu_0^n]_{bid} = \emptyset$ .*

However, as the dangerous places can consume from the same static places, better bounds can be found by solving the proper optimization problem using integer linear programming. To get such a better bound one pays by the fact that the integer linear programming is NP-hard.

**Theorem 4.** *Let  $[m_0]_{dg} \neq \emptyset$  and let  $x : [m_0]_{dg} \rightarrow \mathbb{N}$  be such nonnegative integer solution of the inequation  $\sum_{m \in [m_0]_{dg}} (m_0(s) - m(s)) \cdot x(m) \leq m_0(s)$  for each  $s \in S$ , that maximizes the sum  $\sum_{m \in [m_0]_{dg}} x(m)$ . Denote  $n = \sum_{m \in [m_0]_{dg}} x(m)$ . Then  $[\mu_0]_{bid} = [\mu_0^n]_{bid}$  and  $(PN, m_0)$  is dynamically sound (i.e. has no instance deadlock) iff  $[\mu_0^n]_{bid} = \emptyset$ .*

Finally, the previous results gives the following direct algorithm for the instance deadlock detection without previous computing of a bound for the number of instances. The algorithm has an rcwf-net  $(PN, m_0)$  which is sound for a single instance as an input and decides whether it is instance deadlock free. If not, it returns also the set of all basic deadlocks of  $(PN^{tr}, \mu_0)$  determining the instance deadlocks of  $(PN, m_0)$ . The algorithm computes dangerous markings  $[\mu_0^n]_{dg}$  for  $n$ -bounded transformed nets, starting with  $n = 1$ , and increasing  $n$  until the sets of dangerous markings  $[\mu_0^{n-1}]_{dg}$  and  $[\mu_0^n]_{dg}$  equal. Because  $[\mu_0^{n-1}]_{dg} \subseteq [\mu_0^n]_{dg} \subseteq [\mu_0]_{dg}$  and the number  $[\mu_0]_{dg}$  of dangerous markings of the transformed net is bounded, the algorithm terminates.

### Algorithm 1

1. load an rcwf-net  $(PN, m_0)$  sound for a single instance
2. compute  $[m_0]_{dg}$
3. if  $[m_0]_{dg} = \emptyset$  return "(PN, m<sub>0</sub>) has no instance deadlock"
4. set  $n = 1$
5. compute  $[\mu_0^n]_{dg}$
6. set  $previous_{dg} = [\mu_0^n]_{dg}$
7. set  $n = n + 1$
8. compute  $[\mu_0^n]_{dg}$
9. if  $previous_{dg} = [\mu_0^n]_{dg}$  go to **11**
10. go to **6**
11. compute  $[\mu_0^n]_{bid}$
12. if  $[\mu_0^n]_{bid} = \emptyset$  return "(PN, m<sub>0</sub>) has no instance deadlock"
13. if  $[\mu_0^n]_{bid} \neq \emptyset$  return "(PN, m<sub>0</sub>) has instance deadlocks determined by the basic deadlocks  $[\mu_0^n]_{bid}$ "

We conclude with the following corollary.

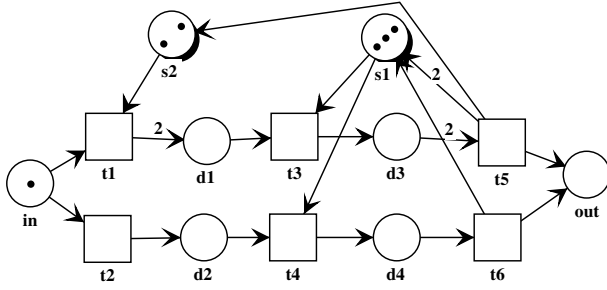


Fig. 10. A corrected minimally restrictive dynamically sound rcwf-net

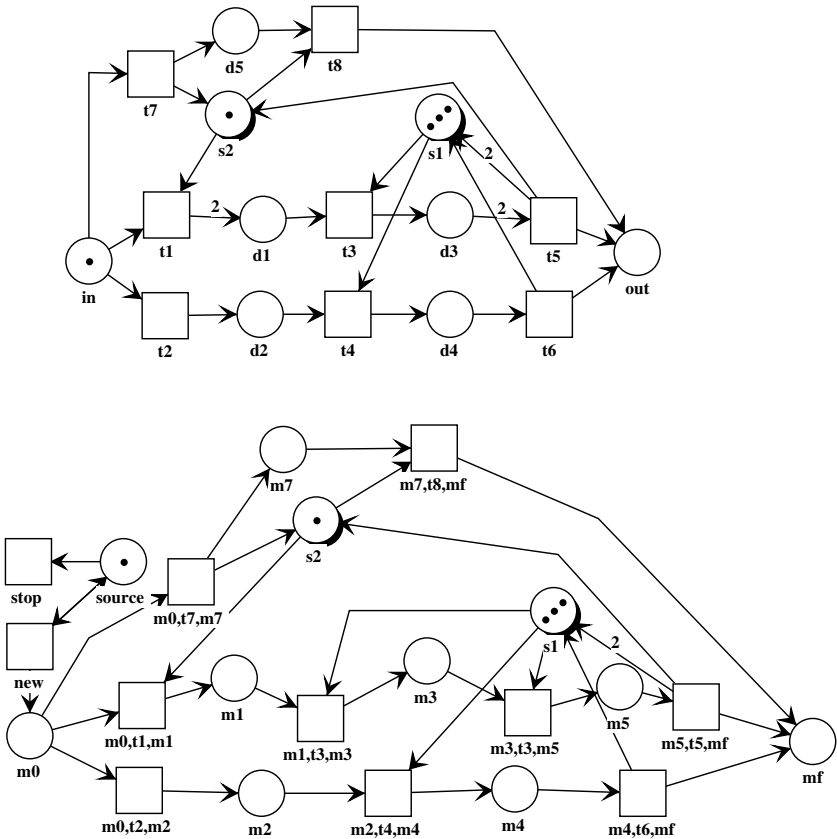
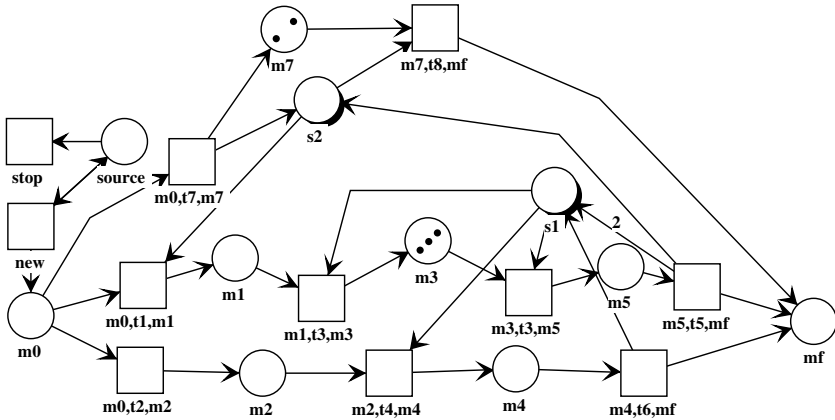


Fig. 11. A net with an unbounded static place and its transformed net

**Corollary 3.** *Testing instance deadlock freeness and dynamic soundness of rcwf-nets is decidable.*





**Fig. 12.** A deadlock situation for 5 instances. No submarking with 3 tokens in the dangerous marking  $m_3$  and no token in other non-static places is reachable.

### 4 Further Research

One line of further research is to find an algorithm constructing a minimally restrictive dynamically sound rcwf-net  $PN_{min}$  for an original rcwf-net  $PN$ , which is not dynamically sound. By minimally restrictive dynamically sound net we mean a net  $PN_{min}$  constructed just by adding static places, such that there is no other net  $PN'$  obtained by adding static places, which is dynamically sound and includes all occurrence sequences of  $PN_{min}$  and at least one sequence which cannot occur in  $PN_{min}$ . An example of such a net for the net from Figure 11 is given in Figure 10.

Another line of future research is to study instance deadlocks in more general setting, relaxing for example boundedness of static places, or allowing different number of tokens in static places in the initial and the final marking, which would correspond to the production and/or the consumption of tokens in static places by single instances. As we illustrate by the net in Figure 11, even relaxing the boundedness of static places causes that many of the results shown in this paper are not longer valid. For example, Theorem 1 is not valid anymore. Figure 12 shows the marking in which the net from Figure 11 deadlocks for 5 instances. However, the net will never deadlock for 3 instances, as no submarking with 3 tokens in the place  $m_3$  and no token in other non-static places is reachable.

### Acknowledgement

Supported by the project APVV-0618-07 of the Slovak Research and Development Agency.

## References

1. van der Aalst, W.M.P., van Hee, K.: *Workflow Management, Models Methods and Systems*. The MIT Press, Cambridge (2002)
2. Esparza, J., Nielsen, M.: Decidability issues for Petri nets—a survey. *J. Inform. Process. Cybernet.* 30(3), 143–160 (1994)
3. van Hee, K.M., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Soundness of Resource-Constrained Workflow Nets. In: Ciardo, G., Darondeau, P. (eds.) *ICATPN 2005*. LNCS, vol. 3536, pp. 250–267. Springer, Heidelberg (2005)
4. Medling, J., van der Aalst, W.M.P.: Errors in the SAP Reference Models. *BPTrends* (June 2006)
5. Desel, J., Juhás, G.: What is a Petri Net? In: Ehrig, H., Juhás, G., Padberg, J., Rozenberg, G. (eds.) *APN 2001*. LNCS, vol. 2128, pp. 1–25. Springer, Heidelberg (2001)
6. Jensen, K.: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use I II III*. Springer, Heidelberg (1997)

# Some Thoughts on Behavioral Programming

*Distinguished Carl Adam Petri Lecture*

David Harel

Dept. of Computer Science and Applied Mathematics  
The Weizmann Institute of Science  
Rehovot 76100, Israel  
dharel@weizmann.ac.il

This talk starts from a dream/vision paper I published in 2008, whose title is a play on that of John Backus' famous Turing Award Lecture (and paper); see [3]. I will propose that — or rather ask whether — programming can be made to be a lot closer to the way humans think about dynamics, and the way they manage to get others (e.g., their children, their employees, etc.) to do what they have in mind. Technically, the question is whether we can liberate programming from its three main straightjackets: (1) having to produce a tangible artifact in some language; (2) having actually to produce *two* separate artifacts (the program and the requirements) and having then to pit one against the other; (3) having to program each piece/part/object of the system separately. The talk will then get a little more technical, providing some modest evidence of feasibility of the dream, via LSCs and the play-in/play-out approach to scenario-based programming [1,2]. The entire body of work around these ideas can be framed as a paradigm that one may term *behavioral programming* [4].

## References

1. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design* 19(1), 45–80 (2001); Ciancarini, P., et al. (eds.): Preliminary version in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS 1999), pp. 293–312. Kluwer, Dordrecht (1999)
2. Harel, D., Marelly, R.: *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, Heidelberg (2003)
3. Harel, D.: Can Programming be Liberated, Period? *IEEE Computer* 41(1), 28–37 (2008)
4. Harel, D., Marron, A., Weiss, G.: *Behavioral Programming* (to appear)

# Petri Nets with Localities and Testing

Jetty Kleijn<sup>1</sup> and Maciej Koutny<sup>2</sup>

<sup>1</sup> LIACS, Leiden University  
Leiden, 2300 RA The Netherlands  
kleijn@liacs.nl

<sup>2</sup> School of Computing Science, Newcastle University  
Newcastle upon Tyne NE1 7RU, United Kingdom  
maciej.koutny@ncl.ac.uk

**Abstract.** In this survey paper, we discuss how to enhance the modelling power of Place/Transition-nets with the notions of ‘locality’ of individual transitions and token ‘testing’ using inhibitor and activator arcs (or, more generally, range arcs). As motivation for these extensions we consider membrane systems – a computational model inspired by the way chemical reactions take place in cells that are divided by membranes into compartments. We explain how key features of membrane systems can be in a natural way captured by transitions with localities (to model compartments) and range arcs (to model inhibitors and promoters). For the resulting model of PTRL-nets, we discuss the synchrony and asynchrony in their behaviours and outline how their causal processes can be defined. Both localities and range arcs render problems, such as boundedness, undecidable in the general case. We therefore present conditions under which one can still decide whether a net is bounded.

**Keywords:** Petri nets, Place/Transition nets, localities, testing, range arcs, GALs systems, membrane systems, causality, processes, barb-events, boundedness, coverability tree.

## 1 Introduction

Extending Petri nets with novel features is a risky endeavour. Usually, it leads to a significant increase of the expressive power and, as a result, existing tools are no longer applicable and important properties become undecidable. On the other hand, when simulating and analysing realistic situations and contexts using Petri nets as a model, it is often absolutely necessary to provide faithful descriptions of complex features which are not directly supported by the standard concepts and notions. This paper focuses on two specific extensions of Petri nets, namely localities allowing to distinguish closely linked components within a system, and testing features allowing one to check for resources without having to claim them for exclusive use.

Petri nets [10,46,47] provide a graphical and mathematically precise framework appropriate for the modelling of distributed systems in which events may happen concurrently and resources may be shared. Place/Transition nets (PT-nets), a typical kind of Petri nets, have places and transitions that are connected

by directed arcs. Places as local states can be marked by a number of tokens to indicate the current availability of resources. Transitions are local actions which when they occur consume tokens from their input places and produce tokens in their output places. Membrane (or P) systems (cf. [42,43,44,45,55]) on the other hand are a computational approach to describe biochemical processes taking place in living cells. The structure of a cell is modelled as a nested structure of membranes delineating its compartments. The reactions which are associated to compartments, are abstracted to rules that specify what new objects are created out of existing ones including possible transfers to neighbouring compartments. Hence there is a direct structural relationship between molecules and rules in a membrane system on the one hand and tokens in places and transitions in a PT-net on the other hand. In particular, potential synchronous or simultaneous execution of reaction rules corresponds to concurrently enabled transitions ([23]). This forms the basis of a translation from membrane systems to PT-nets which faithfully reflects the dependencies and independencies between executions of reaction rules as well as the availability of resources and which thus opens the way to adapting Petri net tools and techniques to the analysis of membrane systems.

To represent compartments, we proposed in [32] to associate localities to transitions (not to places since these — by their very nature — already explicitly support the local aspects of resources). Assigning localities to transitions is relevant in case there is some behavioural significance in the fact that two transitions are co-located or not. Membrane systems are often assumed to evolve in a fully synchronous fashion, which means that at each tick of a global clock as many reaction rules as possible occur, depending on the availability of molecules in the current configuration (the distribution of molecules over the compartments). In this case one would consider for the PT-nets a maximal step semantics (as in [5,19]) in which localities play no role. However, this global synchronicity is not always reasonable from a biological point of view ([43]). It is often more natural to assume that synchronicity is restricted to the compartments with the reactions taking place in synchronised pulses restricted to compartments. In that case, the PT-net would execute under a locally maximal step semantics, i.e., for each locality active in that step, as many transitions as possible are executed. This feature is actually quite common in systems with distributed local control operating under a GALS (globally asynchronous locally synchronous) paradigm, like hardware systems-on-a-chip [52]. Thus localities allow us to define local synchronicity giving rise to a new and interesting locally maximal concurrent semantics for PT-nets.

Another property of chemical reactions which has been considered in the context of membrane systems, is the blocking or triggering of reactions by the presence of certain molecules. Actually, testing for the absence of resources (tokens in places) by means of so-called inhibitor arcs is one of the earliest extensions of the standard Petri net theory, first introduced in [1] in 1973 and discussed in, e.g., [17,34,46]. Other forms of testing come in the form of context [40], activator [18] and read [54] arcs. They all have a slightly different semantical

interpretation, but in essence they test for presence of tokens in specific places. Both kinds of testing have been subsumed and generalised through the concept of range arcs [25] which allow one to test whether the amount of tokens in a place is in-between two given threshold values.

Testing features and locally maximal concurrency (even in case of only one locality, i.e., maximal concurrency) are both proper extensions of the modelling power of PT-nets. Each of them makes it possible to simulate Turing machines and thus lead to undecidable reachability and boundedness problems [5,11,16,50]. Consequently, we have to reconsider and possibly adapt the analysis tools and techniques developed for PT-nets if we want to transfer them to membrane systems and the area of GALs systems in general.

In the remainder of this paper, we focus on PT nets with localities and (local) maximal concurrency issues. Moreover, we will combine them with testing facilities in the form of range arcs. It is our aim not only to define precisely all these notions and concepts, but also to outline to what extent important Petri net approaches can still be applied. In particular, we will look at a process semantics of the resulting net models that should be useful to describe the causality structure of ongoing behaviour, and at adapted coverability graphs that should support the investigation of capacity and boundedness issues. The idea is that we provide a sketch of the progress that has been made. Most of the definitions and results discussed have already been published and are quoted without proofs.

## 2 Preliminaries

This section introduces the notation and terminology used throughout.

A multiset over a set  $X$  is a function  $\mu : X \rightarrow \mathbb{N}$ , and an *extended* multiset a function  $\mu : X \rightarrow \mathbb{N} \cup \{\omega\}$ . We assume that  $\omega + \alpha = \omega - \alpha = k \cdot \omega = \omega$ ,  $n - \omega = 0 \cdot \omega = 0$ , and  $n < \omega$ , for  $n \geq 0$ ,  $k > 0$ , and  $\alpha \in \mathbb{N} \cup \{\omega\}$ . In this paper,  $X$  will always be finite. Clearly, sets can be regarded as special multisets and, in turn, multisets can be regarded as special extended multisets.

For two distinct extended multisets  $\mu$  and  $\mu'$  over  $X$ , we denote  $\mu < \mu'$  if  $\mu(x) \leq \mu'(x)$  for all  $x \in X$ . Moreover,  $\mu \leq \mu'$  if  $\mu < \mu'$  or  $\mu = \mu'$ . The addition  $(\mu + \mu')$  and subtraction  $(\mu - \mu')$  of extended multisets (in the latter case satisfying  $\mu' \leq \mu$ ) as well as multiplication  $(\alpha \cdot \mu)$  of an extended multiset by  $\alpha \in \mathbb{N} \cup \{\omega\}$  are defined point-wise.

If  $\mu$  is a multiset,  $\mu'$  an extended multiset, both over  $X$ , and  $k \in \mathbb{N}$ , then  $\mu \subseteq_k \mu'$  if, for all  $x \in X$ ,  $\mu(x) = \mu'(x)$  if  $\mu'(x) < \omega$ , and otherwise  $\mu(x) > k$ . Intuitively, this means that  $\mu$  approximates the  $\omega$  values in  $\mu'$  with integers greater than  $k$ .

A *Place/Transition net* (or PT-net) is a tuple  $N \stackrel{\text{def}}{=} (P, T, W, M_0)$  such that  $P$  and  $T$  are disjoint finite sets of *places* and *transitions*, respectively;  $W : (T \times P) \cup (P \times T) \rightarrow \mathbb{N}$  is the *weight* function; and  $M_0 \rightarrow \mathbb{N}$  is the *initial* marking. In general, any multiset of places is a *marking*.

In diagrams, places are drawn as circles and transitions as rectangles. The marking  $M(p)$  of a place is indicated by drawing  $M(p)$  small black tokens inside

the circle representing  $p$ . If  $W(x, y) \geq 1$  for some  $(x, y) \in (T \times P) \cup (P \times T)$ , then  $(x, y)$  is an *arc* leading from  $x$  to  $y$ . As usual, arcs are annotated with their weight if this is 2 or more. We assume that, for every  $t \in T$ , there is a place  $p$  such that  $W(p, t) \geq 1$  (i.e., the net is T-restricted).

Given a transition  $t$ , we denote by  $\text{POST}(t)$  the multiset of places given by  $\text{POST}(t)(p) \stackrel{\text{df}}{=} W(t, p)$ , and by  $\text{PRE}(t)$  the multiset of places given by  $\text{PRE}(t)(p) \stackrel{\text{df}}{=} W(p, t)$ . These notations extend to finite multisets  $U$  of transitions in the following way:  $\text{POST}(U) \stackrel{\text{df}}{=} \sum_{t \in U} U(t) \cdot \text{POST}(t)$  and  $\text{PRE}(U) \stackrel{\text{df}}{=} \sum_{t \in U} U(t) \cdot \text{PRE}(t)$  are multisets of places.

We define an execution semantics of PT-nets in terms of concurrently occurring transitions. A *step* of  $N$  is a multiset of transitions,  $U : T \rightarrow \mathbb{N}$ . It is *enabled*, at a marking  $M$  if  $\text{PRE}(U) \leq M$ . An enabled step  $U$  can *executed* leading to the marking  $M' \stackrel{\text{df}}{=} M - \text{PRE}(U) + \text{POST}(U)$ , denoted  $M[U]M'$ . Thus the effect of executing  $U$  is the accumulated effect of executing each of its transitions (taking into account their multiplicities in  $U$ ). Note that the empty step is always enabled and that its execution has no effect, i.e.,  $M' = M$ . A *step sequence* from a marking  $M$  to marking  $M'$  is a possibly empty sequence  $\sigma = U_1 \dots U_n$  of non-empty steps  $U_i$  such that

$$M = M_0 [U_1] M_1 [U_2] M_2 \cdots M_{n-1} [U_n] M_n = M' ,$$

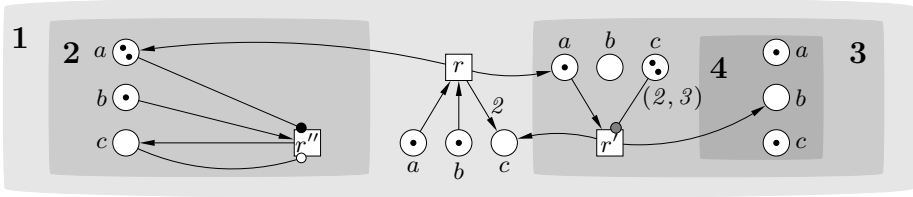
for some markings  $M_1, \dots, M_{n-1}$ . We then denote  $M[\sigma]M'$  and call  $M'$  *reachable* from  $M$ . The set of *reachable* markings of  $N$  is given by  $[M_0] \stackrel{\text{df}}{=} \{M \mid \exists \sigma : M_0[\sigma]M\}$ , and  $N$  is *bounded* if this set is finite.

Finally, the sequential (or interleaving) semantics of  $N$  is obtained by the restriction to singleton steps.

### 3 Extending PT-Nets with Localities and Range Arcs

Membrane systems are a computational model inspired by the compartmentisation and functionality of living cells (cf. [42,43,44,45,55]). Biochemical reactions take place in the compartments of a cell and membrane systems represent these (stoichiometric) reactions through rules specifying what new molecules can be produced from existing ones, and then possibly transferred to neighbouring compartment(s). In its basic form, a membrane system consists of nested membranes which in turn determine compartments, and each distribution of molecules over the compartments determines a configuration. The dynamics of a membrane system derived from its reaction rules, is a form of multiset rewriting [8].

There is an obvious connection between biochemical reactions and net transitions (see, e.g., [49]). This correspondence thus forms a natural basis for a translation from membrane systems to Petri nets (see, e.g., [48]). In [32], also the compartments are taken into account. This is illustrated in Figure 1. Here, the compartments 1–4 are indicated by varying shading. Each kind of molecule ( $a$ ,  $b$ ,  $c$ ) in a compartment is represented by a (labelled counter) place, while each reaction rule is represented by a transition. For example, transition  $r$  represents the rule  $ab \rightarrow cca_{in} 2a_{in} 3$  in compartment 1, which specifies that (in



**Fig. 1.** A PTRL-net modelling a membrane system

compartment 1) two molecules,  $a$  and  $b$ , can react to produce four molecules: two molecules  $c$  remain in the same compartment, one molecule  $a$  is moved to compartment 2, and another molecule  $a$  is moved to 3. Transition  $r'$  represents the more complicated rule  $a \rightarrow c_{out}b_{in}4|_{cc, \neg ccc}$  belonging to compartment 3 which states that molecule  $a$  can split into molecule  $c$  sent out of compartment 3 to the immediately enclosing compartment 1, and molecule  $b$  sent into compartment 4. However, the rule also stipulates that this can happen only if there are in compartment 3, at least 2 molecules  $c$  and, at the same time, no more than 3 molecules  $c$ . This constraint is represented by a special *range* arc [25] between transition  $r'$  and the place for molecules  $c$  in compartment 3. It means that  $r'$  is enabled only if the number of tokens in this place is in the range  $[2, 3]$ . Transition  $r''$  represents the rule  $b \rightarrow c|_{a, \neg c}$  belonging to compartment 2, which changes molecule  $b$  into molecule  $c$  provided that there is at least one  $a$  and no  $c$ 's in compartment 2. These constraints are captured by two arcs, an activator arc and an inhibitor arc, joining  $r''$  with the appropriate molecule counters in compartment 2.

In this way, arcs of various types (with activator and inhibitor arcs as special instances of range arcs) precisely determine the execution of single transitions in a manner which directly reflects the multiset rewriting rules of the membrane system. In turn, these arcs imply dependencies, such as causality and conflict, between executed transitions and hence also between the reaction rules of the original membrane system. By assigning each transition to a unique locality, the compartmentisation of the system is modelled. (It is not necessary to introduce explicit localities for places.)

Membrane systems are often assumed to evolve in a synchronous fashion meaning that as many instances of reaction rules as possible are executed at a time. Thanks to the faithful, structural translation, such behaviour corresponds directly to a maximal step semantics in the associated Petri net. In case, that synchronicity is restricted to compartments (meaning that, for each currently active compartment, as many instances of the reaction rules it comprises as possible are executed at a time), we take advantage of the fact that each transition is assigned to a unique locality and modify the step execution rule to enforce locally maximal parallelism, i.e., for each active locality as many transitions as possible are executed.

Thus, membrane systems are linked in a direct way to PT-nets with localities and range arcs; in particular the transition systems generated are preserved



(modulo isomorphism). As a result, tools and techniques developed for Petri nets can be used (perhaps after a suitable adaptation) to describe, analyse, and verify behavioural properties of membrane systems.

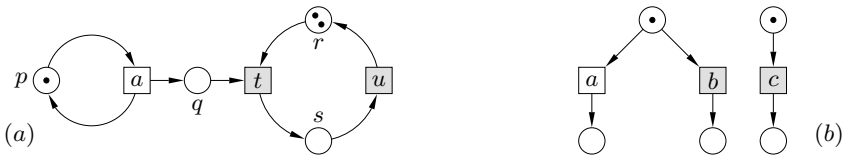
Formally, a *PT-net with range arcs and localities* (or PTRL-net) is a tuple  $NRL \stackrel{\text{def}}{=} (P, T, W, R, L, M_0)$  such that  $(P, T, W, M_0)$  is a PT-net,  $R : P \times T \rightarrow \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$  is the mapping defining *range arcs*, and  $L : T \rightarrow \mathbb{N}$  is the *locality* mapping.

In diagrams, transitions belonging to the same locality either have the same shade or are displayed on a gray background of the same shade. A range arc  $R(p, t)$  is drawn as an arrow from  $p$  to  $t$  with a small gray circle as arrowhead and annotated with  $R(p, t)$ . If  $R(p, t) = (0, \infty)$  then the range arc has no impact on the enabledness of  $t$  and is omitted. If  $R(p, t) = (0, m)$  with  $m \in \mathbb{N}$ , then the range arc is a (weighted) *inhibitor* arc and  $p$  an *inhibitor* place ( $t$  is enabled provided that  $p$  does not contain more than  $m$  tokens); moreover, an arrow is drawn from  $p$  to  $t$  with a small open circle as arrowhead and annotated with the weight  $m$  if  $m > 0$ . If  $R(p, t) = (k, \infty)$  with  $k > 0$ , then the range arc is a (weighted) *activator* arc and  $p$  an *activator* place ( $t$  is enabled provided that  $p$  contains at least  $k$  tokens); moreover, an arrow is drawn from  $p$  to  $t$  with a small black circle as arrowhead and annotated with the weight  $k$  if  $k > 1$ .

Given a marking  $M$ , a step  $U$  is *enabled* if it is enabled in the underlying PT-net and, for every place  $p$  and transition  $t \in U$ , we have  $k \leq M(p) \leq m$ , where  $R(p, t) = (k, m)$ . Moreover,  $U$  is *lmax-enabled* if  $U$  cannot be extended by a transition  $t$  satisfying  $L(t) \in L(U)$  to yield a step which is enabled at  $M$ . Such a definition of enabledness is based on an *a priori* condition: the adherence to range arc constraints is checked *before* the step is executed which seems to be the natural approach for membrane systems. In the *a posteriori* approach (see [7]), one would also require that the range arc constraints are true *after* executing  $U$ . Yet another definition for enabling when activator arcs (or rather read arcs) are involved is given in [54].

In what follows, if we ignore all the aspects relating to range arcs, then  $NRL$  is a PTL-net. Similarly, if we ignore all the aspects relating to the locality mapping then the net is a PTR-net. Moreover, it is a PTI-net if all non-trivial range arcs are inhibitor arcs.

To explain some issues relating to localities, let us consider the PTL-net in Figure 2(a) modelling a system consisting of one producer (modelled by the



**Fig. 2.** A PTL-net  $NL_0$  modelling a one-producer/two-consumers system (a), and a conflict situation between localities (b)

token in  $p$ ) and two co-located consumers (indicated by two tokens in the single place  $r$ ). The buffer-like place  $q$  in the middle holds items produced by the producer using the ‘add’ transition  $a$ , and consumed by the consumers using the ‘take’ transition  $t$ . Transitions  $t$  and  $u$  (for ‘use’) belong to the same locality and  $a$  to another one. Under the PT-net semantics this net could execute the step sequence  $\{a\}\{t\}\{a\}\{t\}$ , but not under the PTL-net lmax-step semantics since after  $\{a\}\{t\}\{a\}$  the step  $\{t, u\}$  comprising two co-located transitions is enabled, violating maximal parallelism w.r.t. the locality of transition  $t$ . A possible legal step sequence is  $\{a\}\{t\}\{a\}\{t, u\}$  as well as  $\{a\}\{t\}\{a\}\{t, u, a\}$ .

Note that steps which are lmax-enabled do not necessarily consist of maximal steps w.r.t. the individual localities. For example, in Figure 2(b),  $\{a, c\}$  is lmax-enabled, but  $\{c\}$  is not lmax-enabled as there is a conflict between transitions coming from two different localities,  $a$  and  $b$ .

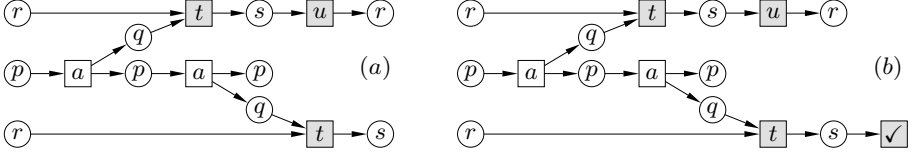
## 4 Processes

Processes of PT-nets [3,4,14,51] are a convenient way of representing concurrent histories by recording explicitly the causal relationships between executed transitions. Each process is a labelled acyclic net, called occurrence net, which through the labelling of its nodes can be seen as an unwinding of the original PT-net along a single concurrent history in which all conflicts between transitions have been resolved. Processes are the basis of a model checking technique based on so-called net unfoldings [13,21,39] as they tend to provide a very compact representation of net behaviour. Moreover, they can be used to directly capture crucial relationships between executed transitions (called events) and places (called conditions) as well as behavioural properties, such as: (i) *causality* which corresponds to directed paths in the process net; (ii) *concurrency* which corresponds to the lack of directed paths between two events; (iii) *reachability*: any maximal set of conditions for which there is no directed path from one to another corresponds to a reachable marking of the original PT-net; and (iv) *executability*: any step sequence from the default initial marking (i.e., exactly one token in each of the conditions without incoming arcs) of the process net defines a step sequence of the original PT-net. The latter criterion is also a proof of consistency of the set-up, as it imposes the condition that any execution of a process corresponds to an execution of the original PT-net.

The paper [22] introduced a general *semantical framework* using which one can define and study in a systematic way process semantics of different classes of Petri nets. The framework links together their step semantics with the process semantics as described above for PT-nets, and describes how processes are to be underpinned by abstract causality structures. In the rest of this section, we will outline the main results of applying the semantical framework to the extensions of the PT-net model we are concerned with in this paper.

*Processes of PTL-nets.* Let us consider the PTL-net  $NL_0$  shown in Figure 2(a) and take one of its step sequences,  $\sigma_0 \stackrel{\text{def}}{=} \{a\}\{a, t\}\{u, t\}$ . Following the standard

process construction developed for PT-nets, we generate from  $\sigma_0$  by unwinding  $NL_0$ , the process net  $\pi_0$  shown in Figure 3(a). However,  $\pi_0$  does not satisfy the important ‘executability’ property, for it allows one to lmax-execute from the default initial marking  $M_0 = \{p, r, r\}$  a step sequence corresponding to step sequence  $\sigma_1 \stackrel{\text{df}}{=} \{a\}\{a\}\{t, t\}\{u\}$  of  $NL_0$ . Clearly,  $\sigma_1$  is legal under the PT-net semantics, but *illegal* under the rules of the PTL-net semantics as the last executed step is not maximal w.r.t. the only active locality.



**Fig. 3.** Two processes constructed for  $NL_0$  and  $\sigma_0$ :  $\pi_0$  in (a) follows the standard PT-net approach, and  $\pi_1$  in (b) follows the PTL-net approach

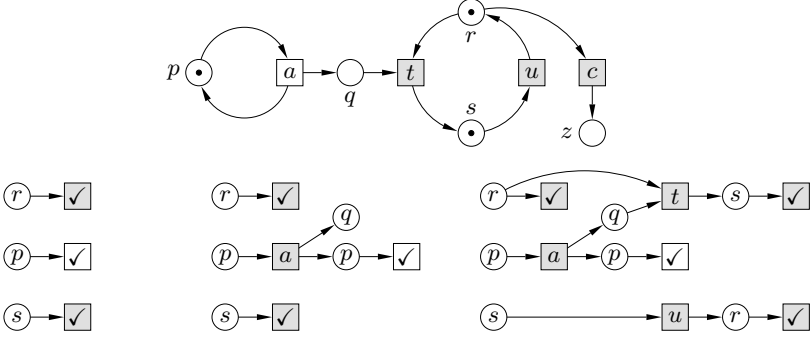
An intuitive reason why  $\pi_0$  is not satisfactory is that it does not record that  $u$  is enabled twice after  $\sigma_2 \stackrel{\text{df}}{=} \{a\}\{a\}\{t, t\}$ , and so the step  $\{u\}$  is not lmax-enabled. To make things work again, [33] proposed to augment the standard construction with information about the presence of *potentially* executable events (including events that were not chosen due to being in conflict with some of the chosen events). Such information comes in the form of special *barb-events*.

For  $NL_0$  and  $\sigma_0$ , a suitable modification is given by the *barb-process*  $\pi_1$  shown in Figure 3(b). It contains a single barb-event labelled with the special symbol  $\checkmark$  and having the same locality and input condition as the ‘missing’ instance of transition  $u$  (output arcs are omitted as barb-events are not meant to be executed). The problem now disappears since after the execution of a step sequence corresponding to  $\sigma_2$ , the step consisting of a single  $u$ -labelled event is not lmax-enabled as it can be extended with the barb-event.

The construction of barb-processes for PTL-nets proceeds as for the standard PT-nets with as only difference that we use barb-events to signal enabledness of transitions from the original PTL-net. In each stage of the construction, already existing and candidate barb-events are considered. Such barb-event is deleted or rejected, respectively, if there is an existing or new event with the same locality whose input conditions are contained in those of the barb-event. This barb-process semantics ‘works’ for the whole class of PTL-nets. In particular, one can show that all step sequences generated by barb-processes correspond to legal step sequences of the original PTL-net.

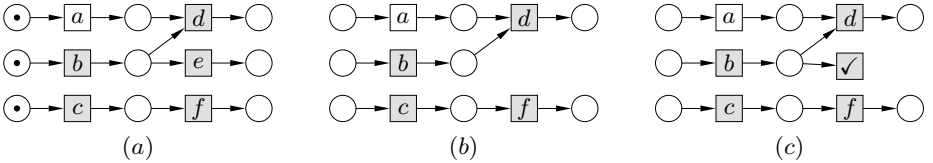
The construction is illustrated in Figure 4 which depicts the generation of a barb-process  $\pi_3$  for the step sequence  $\sigma_3 \stackrel{\text{df}}{=} \{a\}\{u, t\}$  of the PTL-net  $NL_1$  shown there as well. In the first stage, we have three barb-events representing the three transitions which can be included in steps lmax-enabled at the initial marking. In the second stage, one of these barb-events disappears due to the execution of

$a$ , but a new barb-event is added. Note that a candidate barb-event with pre-conditions labelled by  $r$  and  $q$  and the same locality as  $t$  has not been added due to the presence of the topmost barb-event. In the final stage, another barb-event disappears and two new barb-events are added.



**Fig. 4.** A PTL-net  $NL_1$ , and the derivation of the barb-process  $\pi_3$  for  $NL_1$  and  $\sigma_3$

It is worth noting that barb-events are needed also for safe PTL-nets (i.e., PTL-nets whose reachable markings never have more than one token in any place). An example is given in Figure 5.



**Fig. 5.** A PTL-net  $NL_2$  (a); a process net  $\pi_4$  constructed for  $\sigma_4 \stackrel{\text{df}}{=} \{a, b, c\}\{d, f\}$  using the standard PT-net construction (b); and a barb-process  $\pi_5$  for  $\sigma_4$  (c). Note:  $\pi_4$  has a step sequence corresponding to  $\{b, c\}\{a, f\}\{d\}$  which is not allowed by  $NL_2$ . This happens because  $\pi_4$  ‘forgot’ that  $e$  was enabled when  $d$  was selected. The barb-event in  $\pi_5$  rectifies the problem.

*Processes of PTR-nets.* To simplify the presentation, we will only consider a subclass of PTR-nets, called PTRC-nets, such that if  $R(p, t) \in \mathbb{N} \times \mathbb{N}$  then there is another place  $p'$  (a *complement* of  $p$ ) such that  $\text{PRE}(p) = \text{POST}(p')$  and  $\text{POST}(p) = \text{PRE}(p')$ . Thus the total number of tokens in  $p$  and  $p'$  is the same in all the reachable markings. Testing whether  $p$  contains no more than  $m$  tokens can now be replaced by testing for the presence of (at least)  $\beta - m$  tokens in  $p'$ , where  $\beta$  is the total number of tokens in  $p$  and  $p'$  at the initial marking, In this way we can use activator arcs in the processes of PTRC-nets to reflect the constraints implied by the range arcs.

For the PTCR-net  $NR_0$  in Figure 6(a) and its possible step sequence  $\sigma_5 \stackrel{\text{df}}{=} \{u, n\}\{f\}\{t, t, t\}\{c\}$ , the resulting process  $\pi_5$  is shown in Figure 6(b). In this case, there is just one place,  $r$ , which can inhibit the enabledness of transitions ( $n$  and  $c$ ). This place does have a complement place,  $s$ , and in this case the total number of tokens in places  $r$  and  $s$  in all reachable markings is  $\beta = 3$ . The inhibitor arc between  $r$  and  $c$  means that  $c$  is enabled provided that  $r$  contains zero tokens. In the process construction this is replaced by a check that the complement place  $s$  contains  $\beta - 0 = 3$  tokens. Similarly, the range arc between  $r$  and  $n$  implies that  $n$  is enabled provided that  $r$  contains no more than two tokens. In the process construction this is replaced by a check that  $s$  contains at least  $\beta - 2 = 1$  token.

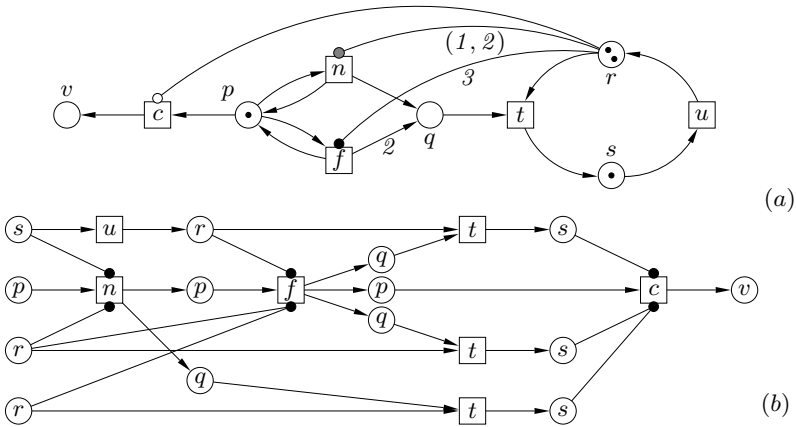


Fig. 6. A PTCR-net  $NR_0$  (a), and a process  $\pi_5$  for  $NR_0$  and  $\sigma_5$  (b).

When one cannot rely on the presence of complements  $p'$  of places  $p$  inhibiting the execution of transitions, another feature is needed to test that such places do not contain too many tokens. The solution introduced in [22,25], was to add ‘on demand’ new artificial conditions to fulfill this role.

*Processes of PTRL-nets.* The general case of a process semantics for PT-nets with localities and weighted inhibitor/activator arcs has been discussed in [26]. This has led to a process semantics combining activator arcs and barb-events and fulfilling the executability criterion: the lmax-step sequences of the processes associated with a PTRL-net all correspond to an lmax-step sequence of that PTRL-net.

*Causality and Concurrency.* The underlying causality structures of processes of PT-nets are (labelled) partial orders. These are defined by the directed acyclic graphs which result by abstracting from the places of processes. Thus each process of a PT-net defines a labelled partial order of events based on production

and consumption of resources and thus reflecting their intrinsic causality and concurrency. Consequently, unorderedness corresponds to independence.

However, representing the causal relations in the behaviour of a PT-net with inhibitor arcs by no more than a partial order is problematic because in the presence of inhibitor arcs, simultaneity and independence are not the same (see [18,22]). To distinguish between them, [18] uses *stratified order structures* consisting of a partial order and an additional weak partial order. This is reflected in the corresponding processes with activator arcs which not only provide information on production and consumption of resources, but also on testing for the presence of tokens. Abstracting from the places now leads to a structure with a partial order defined as before on basis of the ordinary arcs and describing pairs of events of which one should occur before the other. In addition, there is a weak partial order derived from the combination of ordinary and activator arcs which defines a not-later-than relation. For example, in Figure 6, the  $n$ -labelled transition cannot occur later than the  $u$ -labelled transition.

Another related approach to describe the causality and concurrency in Petri net behaviour is provided by *trace theory*. Trace theory has an algebraic, formal language background and as such may contribute new tools and insights. The basic idea relevant here, is that sequential observations that can be associated to a common execution of a Petri net are seen as equivalent. Originally, for Mazurkiewicz traces [38], a binary independence relation is used to define which events do not have to occur in a particular order. Replacing the independence relation by separate notions of simultaneity and serialisability leads to comtraces (equivalence classes of step sequences) which correspond to stratified order structures in the same way as Mazurkiewicz traces correspond to partial orders. Mazurkiewicz traces and comtraces correspond directly to the labelled partial orders defined by the process semantics of subclasses of PT-nets and PTI-nets, respectively (to be precise, Elementary Net Systems and Elementary Net Systems with inhibitor arcs). Extending this trace point of view to more general net classes could be based on their process semantics (see [29] for more details of this approach).

*Reachability.* In ordinary occurrence nets, slices, i.e., maximal sets of incomparable conditions, are exactly those markings which can be reached from the default initial marking. Consequently, with the reachability criterion satisfied by the process semantics, various verification questions, such as marking reachability, can be easily treated using these processes. However, in case of processes with activator arcs (activator occurrence nets), the situation is more complicated (see [22]). First of all one has to take the dependencies induced by the activator arcs into account when defining the concept of a slice. Then, it turns out that even though all slices correspond to configurations reachable from the default initial marking, there may be slices from which the default final marking of the process net is not reachable. This leads in general to more involved proofs.

Still, as pointed out in [25], this process semantics could provide a basis for an efficient verification technique for PTR-nets just like occurrence nets provide a basis for unfolding based model checking for the class of PT-nets (in the

style of, e.g., [13,21,39]). This follows from the observation that when conflicts are included (leading to branching processes with activator arcs similar to the branching processes of [12]), the problem of finding whether a given set of conditions is a marking reachable from the default initial marking is NP-complete.

*Infinite processes.* What we have presented here concerns finite behaviours and structures. Everything, however, can be extended to the infinite case where most of the proofs developed for the finite case still apply after fairly standard additions and modifications. This is essentially due to the well-foundedness of the objects we deal with, making it possible to carry over arguments from the finite case to the infinite case (see [24,25]).

## 5 Coverability

An extended marking  $M$  (i.e., an extended multisets of places) is *covered* by an extended marking  $M'$  if  $M(p) \leq M'(p)$  for every place  $p$ .

Coverability is generally regarded as an important tool for the behavioural analysis of distributed dynamic systems whose states may contain certain kinds of (unbounded) resources. In particular, coverability can be used to verify the boundedness of a PT-net which amounts to saying that there are finitely many reachable markings. This can be done by constructing a ‘coverability tree’ (CT) introduced in [20] and investigated in, e.g., [11,15,46]. Such a tree, rather than giving the exact state space (of reachable markings), provides approximations in terms of extended markings covering reachable markings of PT-nets. CTs can be used to answer also questions related to the boundedness of local states (resources) such as ‘will there be enough resources available?’ (e.g., to avoid deadlocks) or ‘is the amount of resources generated unbounded?’ (hence requiring unlimited capacity of certain parts of the system). CTs can be a tool for deciding behavioural properties even in the case of infinite state spaces as the constructed CT is always finite, with the termination of the construction being based on a ‘monotonicity’ property implying that no behaviour is lost (and can thus be repeated) when more resources become available.

The standard CTs are defined for the sequential (interleaving) semantics of PT-nets and, as a consequence, issues relating to a step-based semantics may not be reflected accurately. To address this issue, [27] considered step coverability trees (SCTs) which directly represent the step sequence semantics of PT-nets. While doing so, [27] introduced extended steps (i.e., extended multisets of transitions) covering executable steps of PT-nets. This allows one, in particular, to use SCTs to answer questions concerned with, e.g., bandwidth which can now be seen as a resource (steps may be unbounded). One may be also be interested in, e.g., whether restricting the throughput can lead to a restricted or even incorrect behaviour.

Crucially, representing (covers of) executable steps seems to be unavoidable if one wants to treat models supporting localities and the (a priori) range testing as both features are inseparable from the notions of steps and step sequence

**Table 1.** An algorithm generating a step coverability tree  $SCT$  of a PT-net  $N$ 


---

$SCT = (V, A, \mu, v_0)$ with initially $V = \{v_0\}$ , $A = \emptyset$ and $\mu[v_0] = M_0$	
$unprocessed = \{v_0\}$	
<b>while</b> $unprocessed \neq \emptyset$	
<b>let</b> $v \in unprocessed$	
<b>if</b> $\mu[v] \notin \mu[V \setminus unprocessed]$ <b>then</b>	[ $v$ not terminal]
<b>for every</b> $M$ and $U$ <b>such that</b> $\mu[v][U]M$ with $U \in select(\mu[v])$	
$V = V \uplus \{w\}$	[add new node]
$A = A \cup \{(v, U, w)\}$	
$unprocessed = unprocessed \cup \{w\}$	
<b>if</b> there is $u$ such that $u \rightsquigarrow v$ and $\mu[u] < M$	(*)
<b>then</b> $\mu[w](p) = (\text{if } \mu[u](p) < M(p) \text{ then } \omega \text{ else } M(p))$	
<b>else</b> $\mu[w] = M$	
$unprocessed = unprocessed \setminus \{v\}$	

---

semantics. Thus SCTs can be regarded as a natural tool for analysing coverability in PTRL-nets, and in the rest of this section we will first present the SCTs and outline results reported in [27,30,31] which can be seen as initial studies in the area of (step) coverability for PTRL-nets. We will look at PTI-nets [27], as well as PTL-nets with one locality [30,31]. In each case, the construction of SCTs will be done for a sub-class of the general model as boundedness is in general undecidable for PTI-nets [16] and PTL-nets with single localities [5].

*Step coverability trees.* The standard CT construction for a PT-net generates extended markings in which the  $\omega$ -entries indicate unboundedness of places. As a consequence, an extended marking may enable infinitely many steps even though a PT-net marking never does so. To build a finite SCT one therefore has to introduce means of representing infinite sets of enabled steps. Following the definition of an extended marking, [27] introduced extended steps and so  $\omega$ -components can label both arcs and nodes of a generated SCT.

Table 1 presents an algorithm for constructing an SCT for a PT-net  $N = (P, T, W, M_0)$  which is a modification of the standard sequential CT construction. The nodes  $V$  and arcs  $A$  are successively added to the initial  $v_0$ . Each node  $v$  is labelled by an extended marking  $\mu[v]$ , and each arc is a triple  $(v, U, w)$ , where  $v, w$  are nodes and  $U$  is an extended step. An extended step  $U$  is enabled at an extended marking  $M$  if  $\text{PRE}(U) \leq M$ , and its execution leads to the extended marking  $(M - \text{PRE}(U)) + \text{POST}(U)$ . The notation  $u \rightsquigarrow v$  indicates that there is a path from node  $u$  to  $v$ . Finally,  $select(M)$  is the set of all extended steps  $U$  enabled at  $M$  with  $U(t) = \omega$  for each transition  $t$  such that the extended step  $\omega \cdot \{t\}$  is enabled at  $M$ .

As shown in [27], the resulting SCT is always *finite* and extends the behavioural information conveyed by the sequential CT, by providing a more concurrency-oriented view of the behaviour of the original PT-net. It satisfies the soundness criteria expected of a coverability tree (see, e.g., [15,11,16]). In



particular, each reachable marking of  $N$  is covered by an extended marking occurring as a label in  $SCT$ , in a ‘tight’ way. That is, for every node  $v$  of  $SCT$  and  $k \geq 0$ , there is a reachable marking  $M$  of  $N$  such that  $M \subseteq_k \mu[v]$ . Thus the  $\omega$ -entries of an extended marking indicate that there are reachable markings of  $N$  which simultaneously grow arbitrarily large on the corresponding places and have, for other places, exactly the same entries as the extended marking. As a result, boundedness of  $N$  can be read off from  $SCT$  by checking that it does not involve any  $\omega$ -entries.

It might not seem to be a good idea to use SCTs for the investigation of properties of PT-nets, as the CTs would in general exhibit a much lower degree of branching. However, this is no longer the case if we consider step based rather than marking based properties. For example, if step  $U$  is enabled at a reachable marking  $M$  of  $N$ , then there is an arc  $(v, W, w)$  in  $SCT$  such that  $M \leq \mu[v]$  and  $U \leq W$ . Moreover, for every  $k \geq 0$  and every  $W$  labelling an arc in  $SCT$ , there is a step  $U$  enabled at a reachable marking of  $N$  satisfying  $U \subseteq_k W$ .

A key reason why the construction of SCT for PT-nets works is that their semantics is monotonic in the sense that if a step  $U$  is enabled at a marking  $M$  and  $M'$  is a marking satisfying  $M < M'$  then  $U$  is also enabled at  $M'$ . This key property no longer holds when we add localities or range arcs.

*PTL-nets with single localities.* PTL-net semantics can be viewed as monotonic in the weaker sense that adding resources can enable larger steps without invalidating already enabled transitions. This weak form of monotonicity was used in [30,31] to construct SCTs for PTL-nets with single localities which, in essence, are PT-nets executed under the maximally concurrent execution rule [5].

Let us assume that  $N$  is a PT-net executed under the maximally concurrent rule. The problem is to find criteria for terminating the development of a coverability tree along an infinite path from the root. To achieve the desired effect, [30,31] introduced a criterion based on a repeated execution of the same sequence of steps. This key result states that  $\kappa\tau, \kappa\tau\tau, \kappa\tau\tau\tau, \dots$  are step sequences of  $N$  iff the following hold (below  $\tau = U_1 \dots U_n$ , and  $M^i$  is the marking reached after executing  $\kappa U_1 \dots U_i$ ):

- $\kappa\tau\tau$  is a step sequence of  $N$ ;
- executing  $\tau$  can only add tokens to places; and
- for all transitions  $t$  and  $i < n$ , there is a place  $p$  to which the sequence of steps  $\tau$  does not add any tokens, and which stops the enabledness of the step  $\{t\} + U_{i+1}$  at  $M^i$  (i.e.,  $M^i(p) - \text{PRE}(U)(p) < W(p, t)$ ).

Thus two repetitions of the same sequence of steps  $\tau$  can be enough to infer an infinite repetition of  $\tau$ . The above is a sufficient but not necessary condition characterising situations which may lead to unboundedness, and to make it also a necessary condition [30,31] restricted the class of PT-nets underlying  $N$ . The idea was to define a class of PTL-nets modelling acyclic networks of finite state components communicating by means of buffered channels.

Individual components in [31] are live and bounded PT-nets such that each reachable (in the PT-net sense) marking enables a unique maximally concurrent

step and, for all cycles in the reachability graph, the relative execution rates of the transitions are constant. (The weighted strongly connected marked graphs [53] used in [30] satisfy these conditions.) The PTL-nets for which [31] develops SCTs are then obtained by taking any number of individual components and connecting them in a network-acyclic manner using arbitrarily many additional places and transitions. The key behavioural property of the resulting PT-nets is that when executed under the maximally concurrent rule, their individual components eventually synchronise, and the net as a whole assumes a cyclic behaviour. More precisely, the step sequences of  $N$  are finite prefixes of a certain infinite sequence of steps of the form  $\kappa\tau\tau\tau\dots$ .

The algorithm in Table I needs to be amended in the following way. First, the mapping  $select(\cdot)$  is as before, but it only takes into account (extended) steps enabled by the maximally concurrent rule. Then the line (\*) is replaced by:

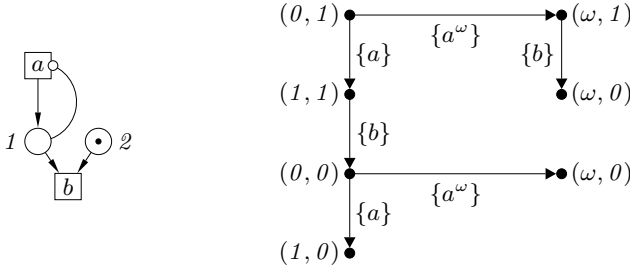
- if there are  $u, u'$  and  $\tau = U_1 \dots U_n$  such that:
- $u \rightsquigarrow u' \rightsquigarrow v$  and  $\mu[u] < \mu[u'] < M$ ;
  - $\tau$  labels the paths from  $u$  to  $u'$  and from  $u'$  to  $v$ ; and
  - for all transitions  $t$  and  $i < n$ , there is a place  $p$  to which the sequence of steps  $\tau$  does not add any tokens, and which stops the enabledness of the step  $\{t\} + U_{i+1}$  at the marking labelling the node reached from  $u$  following the arcs labelled by  $U_1 \dots U_i$

Thus, rather than an inequality on markings, *double* inequalities with the *same* maximal step sequences in-between are needed together with the additional condition guaranteeing the repeatability of that step sequence. One then can then show that the algorithm always terminates and the expected soundness criteria do hold.

*Primitive PTI-nets.* Inhibitor arcs destroy completely the monotonicity in net executions which, intuitively, makes PTI-nets even more difficult to handle than PTL-nets. To ensure the termination of the coverability construction in the sequential case, [6] introduced ‘primitive’ PTI-nets which include PT-nets yet still have more expressive power. Intuitively, when an inhibitor place in such a PTI-net contains more than a certain threshold of tokens (its emptiness limit), no transition which tests it for emptiness can occur anymore.

The class of primitive PTI-nets was reconsidered in [27], but this time under the *a priori* step sequence semantics. It is important to emphasize at this point that for PTI-nets executed under the *a priori* step sequence semantics, marking reachability cannot be reduced to marking reachability under the sequential semantics. This provided an additional motivation to develop SCTs for PTI-nets.

A PTI-net  $N = (P, T, W, R, M_0)$  with unweighted ordinary arcs is *primitive* if there is an integer  $EL$  (the ‘emptiness limit’) such that for every reachable marking  $M$  and every inhibitor place  $p$ , if  $M(p) > EL$  then for every marking  $M'$  reachable from  $M$  and transition  $t$  enabled at  $M'$ , it is the case that  $M'(p) > m$ , where  $R(p, t) = (0, m)$ .



**Fig. 7.** A primitive PTI-net with  $EL = 1$  and its step coverability tree. Note that the pair  $(x, y)$  labelling a node of the latter represents an extended marking  $M$  such that  $M(1) = x$  and  $M(2) = y$ .

To work as required for a primitive PTI-net  $N$ , the algorithm in Table [□](#) needs to be instantiated with a new *select*(.) function. In this case,  $select(\mu[v])$  is the set of all extended steps of transitions  $U$  enabled at  $\mu[v]$  such that  $U(t) \in \{0, 1, \dots, EL, \omega\}$ , for each transition  $t$  such that  $\omega \cdot \{t\}$  is enabled at  $\mu[v]$ . Moreover, to compare extended markings,  $<$  is replaced by another ordering  $\sqsubset$  such that for any two distinct extended markings,  $M$  and  $M'$ , we have  $M \sqsubset M'$  if  $M(p) \leq M'(p)$ , for all places  $p$ , and  $M(p) = M'(p)$  for all inhibitor places  $p$ , whenever  $M(p) \leq EL$ . Figure [□](#) shows an example application of the modified construction.

Intuitively,  $select(\mu[v])$  is defined in such a way that if a non-selected extended step enabled at  $\mu[v]$  inserts some tokens into an inhibitor place  $p$ , then it necessarily inserts at least  $EL+1$  tokens, making from this point on the inhibiting features of  $p$  void. And the step itself will be covered by at least one step in  $select(\mu[v])$ . The ordering  $\sqsubset$  was introduced in [□](#) and is intended to ensure that inhibitor places are treated as such (and their marking is not replaced by  $\omega$ 's) until the threshold value  $EL$  has been passed. One then can then show that the algorithm always terminates and the expected soundness criteria still hold.

## 6 Further Issues

The model of PTRL-nets discussed in this paper has wider applicability within the general field of membrane systems, in particular, if one allows reaction rules which can dissolve and/or thicken the membranes. In such a case, a membrane system may dynamically change its structure due to the execution of specific reaction rules. As shown in [□](#), the extended model of membrane structures can still be treated using behaviourally equivalent PTRL-nets. In this case, inhibitor and activator arcs are the key devices allowing one to model the change of the membrane structure i.e., they are used to implement control structures in the model.

So far we have discussed ways of ensuring correctness of systems with localities through the behavioural analysis of the corresponding PTRL-nets. An alternative approach aimed at ensuring correctness is to provide means to construct

such systems through automated synthesis from behavioural specifications. Papers such as [35,36] considered a particular instance of this approach which aims at constructing PTRL-nets from specifications given in terms of a finite transition systems with arcs labelled by steps of executed transitions (with or without knowing the localities of individual transitions).

One might argue that the expressiveness of PTRL-nets is somewhat constrained by the fact that each transition belongs to a unique locality, and so the localities are all *non-overlapping*. In [37] this assumption has been dropped resulting in a net model which could provide a greater scope for faithful (or direct) modelling features implied by the complex nature of, for example, modern VLSI systems or biological systems. For such an extended model, it is again possible to automatically construct nets from finite step transition systems. Note that [35,36,37] proposed solutions to the synthesis problem using the notion of a region of a transition system (see, e.g., [241]) and the notion of a step firing policy [9].

## 7 Concluding Remarks

Petri nets with localities and range arcs are a model capable of modelling a wide range of increasingly sophisticated man-made systems as well as biological processes which occur in living tissue and the development of organisms. Already, the PTRL-net model offers a flexible modelling technique in which, for example, membrane systems with dynamic structure can be specified and analysed.

We feel that there are two major directions for future research in this area. Firstly, there is the continuation of the investigation into and the development of implementable efficient analytical techniques, for instance, based on the process semantics or step coverability trees. Secondly, it seems worthwhile to investigate the possibilities of a further, careful enhancement of the modelling features of PTRL-nets, such as the introduction of overlapping localities (which perhaps then should be called *vicinities*), and testing the new extended net model using representative case studies.

## Acknowledgements

This research was supported by the RAE&EPSRC DAVAC, EPSRC VERDAD and CASINO and EU RODIN projects.

## References

1. Agerwala, T.: A Complete Model for Representing the Coordination of Asynchronous Processes. Hopkins Computer Research Report 32, Johns Hopkins University (1974)
2. Badouel, E., Darondeau, P.: Theory of Regions. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
3. Best, E., Devillers, R.: Sequential and Concurrent Behaviour in Petri Net Theory. Theoretical Computer Science 55, 87–136 (1988)

4. Best, E., Fernández, C.: Nonsequential Processes. A Petri Net View. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (1988)
5. Burkhard, H.-D.: On Priorities of Parallelism: Petri Nets Under the Maximum Firing Strategy. In: Salwicki, A. (ed.) Logic of Programs 1980. LNCS, vol. 148, pp. 86–97. Springer, Heidelberg (1983)
6. Busi, N.: Analysis Issues in Petri Nets with Inhibitor Arcs. Theoretical Computer Science 275, 127–177 (2002)
7. Busi, N., Pinna, G.M.: Process Semantics for Place/transition Nets with Inhibitor and Read Arcs. Fundamenta Informaticae 40, 165–197 (1999)
8. Calude, C.S., Păun, G., Rozenberg, G., Salomaa, A. (eds.): Multiset Processing. LNCS, vol. 2235. Springer, Heidelberg (2001)
9. Darondeau, P., Koutny, M., Pietkiewicz-Koutny, M., Yakovlev, A.: Synthesis of Nets with Step Firing Policies. Fundamenta Informaticae 94, 275–303 (2009)
10. Desel, J., Reisig, W., Rozenberg, G. (eds.): Lectures on Concurrency and Petri Nets. LNCS, vol. 3098. Springer, Heidelberg (2004)
11. Desel, J., Reisig, W.: Place/Transition Petri Nets. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 122–173. Springer, Heidelberg (1998)
12. Engelfriet, J.: Branching processes of Petri nets. Acta Informatica 28, 575–591 (1991)
13. Esparza, J., Heljanko, K.: Unfoldings: A Partial-order Approach To Model Checking. Springer, Heidelberg (2008)
14. Goltz, U., Reisig, W.: The Non-sequential Behaviour of Petri Nets. Information and Control 57, 125–147 (1983)
15. Hack, M.: Decision Problems for Petri Nets and Vector Addition Systems. Technical Memo 59, Project MAC, MIT (1975)
16. Hack, M.: Petri Net Languages. Technical Report 159, MIT (1976)
17. Hack, M.: Decidability Questions for Petri Nets. PhD Thesis, MIT (1976)
18. Janicki, R., Koutny, M.: Semantics of Inhibitor Nets. Information and Computation 123, 1–16 (1995)
19. Janicki, R., Lauer, P.E., Koutny, M., Devillers, R.: Concurrent and Maximally Concurrent Evolution of Nonsequential Systems. Theoretical Computer Science 43, 213–238 (1986)
20. Karp, R.M., Miller, R.E.: Parallel Program Schemata. J. Comput. Syst. Sci. 3, 147–195 (1969)
21. Khomenko, V., Kondratyev, A., Koutny, M., Vogler, W.: Merged Processes: a New Condensed Representation of Petri Net Behaviour. Acta Informatica 43, 307–330 (2006)
22. Kleijn, H.C.M., Koutny, M.: Process Semantics of General Inhibitor Nets. Information and Computation 190, 18–69 (2004)
23. Kleijn, J., Koutny, M.: Synchrony and Asynchrony in Membrane Systems. In: Hoogeboom, H.J., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2006. LNCS, vol. 4361, pp. 66–85. Springer, Heidelberg (2006)
24. Kleijn, H.C.M., Koutny, M.: Infinite Process Semantics of Inhibitor Nets. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 282–301. Springer, Heidelberg (2006)
25. Kleijn, J., Koutny, M.: Processes of Petri Nets with Range Testing. Fundamenta Informaticae 80, 199–219 (2007)
26. Kleijn, J., Koutny, M.: Processes of Membrane systems with Promoters and Inhibitors. Theoretical Computer Science 404, 112–126 (2008)

27. Kleijn, J., Koutny, M.: Steps and Coverability in Inhibitor Nets. In: Lodaya, K., Mukund, M., Ramanujam, R. (eds.) *Perspectives in Concurrency Theory*, pp. 264–295. Universities Press, Hyderabad (2008)
28. Kleijn, J., Koutny, M.: A Petri Net Model for Membrane Systems with Dynamic Structure. *Natural Computing* 8, 781–796 (2009)
29. Kleijn, J., Koutny, M.: Formal Languages and Concurrent Behaviours. In: Bel-Enguix, G., Dolores Jiménez-López, M., Martín-Vide, C. (eds.) *New Developments in Formal Languages and Applications*, pp. 125–182. Springer, Heidelberg (2008)
30. Kleijn, J., Koutny, M.: Applying Step Coverability Trees to Communicating Component-Based Systems. In: Sirjani, M. (ed.) *FSEN 2009. LNCS*, vol. 5961, pp. 178–193. Springer, Heidelberg (2010)
31. Kleijn, J., Koutny, M.: Step Coverability Algorithms for Communicating Systems. To appear in *Science of Computer Programming* (2010)
32. Kleijn, J., Koutny, M., Rozenberg, G.: Towards a Petri Net Semantics for Membrane Systems. In: Freund, R., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *WMC 2005. LNCS*, vol. 3850, pp. 292–309. Springer, Heidelberg (2006)
33. Kleijn, J., Koutny, M., Rozenberg, G.: Process Semantics for Membrane Systems. *Journal of Automata, Languages and Combinatorics* 11, 321–340 (2006)
34. Kosaraju, S.R.: Limitations of Dijkstra’s Semaphore Primitives and Petri Nets. *Operating Systems Review* 7, 122–126 (1973)
35. Koutny, M., Pietkiewicz-Koutny, M.: Synthesis of Elementary Net Systems with Context Arcs and Localities. *Fundamenta Informaticae* 88, 307–328 (2008)
36. Koutny, M., Pietkiewicz-Koutny, M.: Synthesis of Petri Nets with Localities. *Scientific Annals of Computer Science* 19, 1–23 (2009)
37. Koutny, M., Pietkiewicz-Koutny, M.: Synthesis of General Petri Nets with Localities. Report CS-TR-1195, School of Computing Science, Newcastle University (2010)
38. Mazurkiewicz, A.W.: Trace Theory. In: Rozenberg, G. (ed.) *APN 1987. LNCS*, vol. 266, pp. 279–324. Springer, Heidelberg (1987)
39. McMillan, K.L.: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. In: Probst, D.K., von Bochmann, G. (eds.) *CAV 1992. LNCS*, vol. 663, pp. 164–174. Springer, Heidelberg (1993)
40. Montanari, U., Rossi, F.: Contextual Nets. *Acta Informatica* 32, 545–586 (1995)
41. Nielsen, M., Rozenberg, G., Thiagarajan, P.S.: Elementary Transition Systems. *Theoretical Computer Science* 96, 3–33 (1992)
42. Păun, G.: Computing with Membranes. *Journal of Computer and System Sciences* 61, 108–143 (2000)
43. Păun, G.: *Membrane Computing. An Introduction*. Springer, Heidelberg (2002)
44. Păun, G., Rozenberg, G.: *A Guide to membrane computing*. *Theoretical Computer Science* 287, 73–100 (2002)
45. Păun, G., Rozenberg, G., Salomaa, A. (eds.): *The Oxford Handbook of Membrane Computing*. Oxford University Press, Oxford (2010)
46. Peterson, J.L.: *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs (1981)
47. Petri, C.A.: Fundamentals of a Theory of Asynchronous Information Flow. In: *Proc. of IFIP Congress 1962*, pp. 386–390. North Holland, Amsterdam (1962)
48. Qi, Z., You, J., Mao, H.: P Systems and Petri Nets. In: Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *WMC 2003. LNCS*, vol. 2933, pp. 286–303. Springer, Heidelberg (2004)
49. Reddy, V.N., Liebman, M.N., Mavrouniotis, M.L.: Qualitative Analysis of Biochemical Reaction Systems. *Comput. Biol. Med.* 26, 9–24 (1996)

50. Reinhardt, K.: Reachability in Petri Nets with Inhibitor Arcs. Technical Report WSI-96-30, Wilhelm Schickard Institut für Informatik, Universität Tübingen (1996)
51. Rozenberg, G., Engelfriet, J.: Elementary Net Systems. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 12–121. Springer, Heidelberg (1998)
52. Stahl, C., Reisig, W., Krstić, M.: Hazard Detection in a GALS Wrapper: A Case Study. In: ACSD 2005, pp. 234–243. IEEE Computer Society, Los Alamitos (2005)
53. Teruel, E., Chrzastowski-Wachtel, P., Colom, J.M., Silva, M.: On Weighted T-Systems. In: Jensen, K. (ed.) ICATPN 1992. LNCS, vol. 616, pp. 348–367. Springer, Heidelberg (1992)
54. Vogler, W.: Partial Order Semantics and Read Arcs. *Theoretical Computer Science* 286, 33–63 (2002)
55. Membrane systems web page, <http://ppage.psystems.eu/>

# A Perspective on Explicit State Space Exploration of Coloured Petri Nets: Past, Present, and Future

Lars M. Kristensen

Department of Computer Engineering  
Bergen University College, Norway  
`lmkr@hib.no`

**Abstract.** We provide a chronological research perspective on the development and application of methods and supporting computer tools for state space exploration and model checking of Coloured Petri Nets. We discuss how the lessons learned from practical applications have influenced current and envisioned future research directions concentrating on the ongoing development of the ASAP state space exploration platform.

Explicit state space exploration is one of the main approaches to model-based verification of concurrent systems and it has been one of the most successfully applied [8] analysis methods for Coloured Petri Nets (CPNs) [10,13,14]. Our work on the development and application of state space methods for CPNs and their supporting computer tools has spanned several years, and it has included the development of three generations of computer tools. State space methods has also generally been a highly active area of research resulting in a vast variety of storage techniques, verification algorithms, and computer tools. The large suite of state space methods available today combined with the power of modern computing platforms allows for the validation of industrial-sized concurrent systems – despite the inherent presence of the state explosion problem [27].

A fundamental guideline governing our research approach has been the development of state space methods that supports the complete CPN modelling language. In particular this means that we do not rely on restrictions on the net structure or inscription language, nor on unfolding to the underlying low-level Petri net. Most state space methods and model checking techniques can be formulated at the level of transitions systems, and are hence transferable between modelling languages. The rich set of data types and associated inscription language which are fundamental building blocks of the CPN modelling language however pose specific challenges for state space methods in the context of CPNs.

Early work concentrated on the development of computer tool support for full state space exploration [2,24] and initial experiments with the equivalence [11], symmetry [4,5,9,12], and the stubborn set methods [26,27]. The symmetry method exploits inherent symmetries in systems to compute a condensed state space where each node represents an equivalence class of states and each arc represents an



equivalence class of events. The equivalence method is a generalisation of the symmetry method in which there is no requirement on the origin of the equivalence relations on states and events. Both of these methods showed difficult to apply in practice [16,15,21] as they require a manual soundness proof for the user-provided symmetry (equivalence). Furthermore, applications of the symmetry method showed that the time required to compute canonical representatives of equivalence classes was excessive – even with the use of advanced group algebraic techniques [22]. The stubborn set method analyses the dependencies between transitions to explore only a subset of the full state space while preserving enough behaviour to answer the verification question being considered. Computing stubborn sets of CPNs requires in worst case time proportional to the size of the underlying low-level Petri net [20]. Hence, restrictions on the modelling language is required to apply the stubborn set method without relying on unfolding.

The difficulties with the practical application of the symmetry, equivalence, and stubborn set methods in the context of CPNs prompted a change in research direction towards methods that aim at making more economical use of memory resources when exploring the ordinary state space. Memory is (in most cases) the limiting factor in state space exploration of CPN models due to the large state vectors. This work resulted in the development of the sweep-line method [3] and the comback method [30,7]. The sweep-line suite of methods [3,18,17,11,23] is aimed at on-the-fly verification and exploits a notion of progress found in many concurrent systems. Exploiting progress allows for the deletion of states from memory during a progress-first traversal of the state space. This in turn reduces peak memory usage. The comback method can be viewed as an exploration-order independent storage mechanism based on hash compaction [25,31]. It allows the usually large state vectors of CPN models to be stored in compact form, and the full state vector of a state is reconstructed when needed for comparison with newly generated states. Unlike the classical hash compaction method, the comback method guarantees full coverage of the state space.

Ongoing work has concentrated on the development of the ASAP state space exploration platform [28] which is intended to constitute the next generation of computer tool support for state space exploration of CPN models. The design of the ASAP platform takes into account many of the lessons learned through the development and application of state space methods outlined above. An important vision of ASAP is to provide the user with coherent support for a large suite of state space methods. ASAP relies on a graphical language for specification of verification jobs allowing users to work on different abstraction levels when applying state space methods. Furthermore, ASAP has a software architecture [29] that allows researchers to extend the tool with new state space methods and have these integrated as first class citizens in the tool. Recent work has also included investigations of state space partitioning schemes [6] targeting distributed [19] and external memory state space exploration of CPN models.

**Acknowledgements.** The work on state space methods for CPNs has involved research collaboration with several co-authors and colleagues. The author is grateful for the contributions of: Jonathan Billington, Gerth S. Brodal, Søren

Christensen, Paul Fleischer, Louise Elgaard, Sami Evangelista, Guy Gallasch, Kurt Jensen, Jens B. Jørgensen, Mads K. Kjeldsen, Charles Lakos, Thomas Mailund, Laure Petrucci, Surayya Urazimbetova, Antti Valmari, Michael Westergaard, Karsten Wolf, and Lin Zhang.

## References

1. Billington, J., Gallasch, G., Kristensen, L.M., Mailund, T.: Exploiting Equivalence Reduction and the Sweep-Line Method for Detecting Terminal States. *IEEE Transactions on Systems, Man, and Cybernetics. Part A: Systems and Humans* 34(1), 23–38 (2004)
2. Christensen, S., Jørgensen, J.B., Kristensen, L.M.: Design/CPN – A Computer Tool for Coloured Petri Nets. In: Brinksma, E. (ed.) *TACAS 1997*. LNCS, vol. 1217, pp. 209–223. Springer, Heidelberg (1997)
3. Christensen, S., Kristensen, L.M., Mailund, T.: A Sweep-Line Method for State Space Exploration. In: Margaria, T., Yi, W. (eds.) *TACAS 2001*. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
4. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting Symmetries in Temporal Logic Model Checking. *Formal Methods in System Design* 9, 77–104 (1996)
5. Emerson, E.A., Sistla, A.P.: Symmetry and Model Checking. *Formal Methods in System Design* 9, 105–131 (1996)
6. Evangelista, S., Kristensen, L.M.: Dynamic State Space Partitioning for External Memory Model Checking. In: Alpuente, M. (ed.) *FMICS 2009*. LNCS, vol. 5825, pp. 70–85. Springer, Heidelberg (2009)
7. Evangelista, S., Westergaard, M., Kristensen, L.M.: The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. *Transactions on Petri Nets and Other Models of Concurrency* 3, 189–215 (2009)
8. Examples of Industrial Use of CP-Nets,  
<http://www.cs.au.dk/CPnets/intro/example/indu.html>
9. Ip, C.N., Dill, D.L.: Better Verification Through Symmetry. *Formal Methods in System Design* 9, 41–75 (1996)
10. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Basic Concepts. Monographs in Theoretical Computer Science, vol. 1. Springer, Heidelberg (1992)
11. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Analysis Methods. Monographs in Theoretical Computer Science, vol. 2. Springer, Heidelberg (1994)
12. Jensen, K.: Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design* 9, 7–40 (1996)
13. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
14. Jensen, K., Kristensen, L.M., 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(3-4), 213–254 (2007)
15. Jørgensen, J.B., Kristensen, L.M.: Computer Aided Verification of Lamports Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. *IEEE Transactions on Parallel and Distributed Systems* 10(7), 714–732 (1999)

16. Jørgensen, J.B., Kristensen, L.M.: Verification of Coloured Petri Nets Using State Spaces with Equivalence Classes. In: Petri Net Approaches for Modelling and Validation, ch. 2. LINCOS Studies in Computer Science, vol. 1, pp. 17–34. Lincoln Europa (2003)
17. Kristensen, L.M., Mailund, T.: A Generalised Sweep-Line Method for Safety Properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002)
18. Kristensen, L.M., Mailund, T.: Efficient Path Finding with the Sweep-Line Method using External Storage. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 319–337. Springer, Heidelberg (2003)
19. Kristensen, L.M., Petrucci, L.: An Approach to Distributed State Space Exploration for Coloured Petri Nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 474–483. Springer, Heidelberg (2004)
20. Kristensen, L.M., Valmari, A.: Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 104–123. Springer, Heidelberg (1998)
21. Lorentsen, L., Kristensen, L.M.: Modelling and Analysis of a Danfoss Flowmeter System using Coloured Petri Nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 346–366. Springer, Heidelberg (2000)
22. Lorentsen, L., Kristensen, L.M.: Exploiting Stabilizers and Parallelism in State Space Generation with the Symmetry Method. In: Proc. of ICACSD 2001, pp. 211–220. IEEE Computer Society, Los Alamitos (2001)
23. Mailund, T.: Analysing Infinite-State Systems by Combining Equivalence Reduction and the Sweep-Line Method. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 314–333. Springer, Heidelberg (2002)
24. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 450–462. Springer, Heidelberg (2003), <http://www.cs.au.dk/CPNTools>
25. Stern, U., Dill, D.L.: Improved Probabilistic Verification by Hash Compaction. In: Camurati, P.E., Eueking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995)
26. Valmari, A.: Stubborn Sets of Coloured Petri Nets. In: Proc. of ICATPN 1991, pp. 102–121 (1991)
27. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
28. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An Extensible Platform for State Space Analysis. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 303–312. Springer, Heidelberg (2009), <http://www.daimi.au.dk/~ascoveco/download.html>
29. Westergaard, M., Kristensen, L.M.: The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 313–322. Springer, Heidelberg (2009), <http://www.daimi.au.dk/~ascoveco/accesscpn/>
30. Westergaard, M., Kristensen, L.M., Brodal, G.S., Arge, L.A.: The ComBack Method – Extending Hash Compaction with Backtracking. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 445–464. Springer, Heidelberg (2007)
31. Wolper, P., Leroy, D.: Reliable Hashing without Collision Detection. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)

# Can Stubborn Sets Be Optimal?

Antti Valmari and Henri Hansen

Tampere University of Technology, Department of Software Systems  
PO Box 553, FI-33101 Tampere, Finland  
{antti.valmari,henri.hansen}@tut.fi

**Abstract.** Literature on the stubborn set and similar state space reduction methods presents numerous seemingly ad-hoc conditions for selecting the transitions that are investigated in the current state. There are good reasons to believe that the choice between them has a significant effect on reduction results, but not much has been published on this topic. This article presents theoretical results and examples that aim at shedding light on the issue. Because the topic is extensive, we only consider the detection of deadlocks. We distinguish between different places where choices can be made and investigate their effects. It is usually impractical to aim at choices that are “best” in some sense. However, one non-trivial practical optimality result is proven.

**Keywords:** Partial order verification, stubborn sets.

## 1 Introduction

In this article we investigate methods for constructing a subset of the state space of a concurrent system in such a way that the deadlocks of the system are preserved. That is, every deadlock of the system is present in the reduced state space, and every deadlock of the reduced state space is a deadlock of the system. We only investigate methods where, when constructing the immediate successor states of a state in the reduced state space, a subset of the (structural) transitions of the system is computed and only the transitions in it are used for computing the successor states.

Consider a concurrent program. If some process is ready to execute a statement that does not in any way depend on or interact with other processes and non-local variables, then it is intuitively obvious that no deadlocks are lost if only that statement is investigated in the current state. This observation is so obvious that it seems impossible to find out who made it first and when. However, its potential for preventing state explosion is limited.

More complicated but also more powerful methods of that kind have been published since 1988, such as *stubborn sets* [10], *persistent sets* [5], and *ample sets* [7]. Many variants of them have been developed in at least three dimensions: the set of preserved properties has varied from just deadlocks to full-fledged stuttering-insensitive temporal logics; different notions of dependency or

interaction between transitions have been used; and different methods for constructing the sets have been proposed. In this article we restrict our attention to deadlocks but analyse the variability in the other two dimensions.

Many methods are available even if the set of preserved properties is restricted to deadlocks. Although they are based on the same overall principles, their details vary. Little is known on their relative reduction power. It can be investigated experimentally or theoretically. Some experimental results made with a computer were presented in [4]. In this article we tackle the issue theoretically and with examples.

Detecting deadlocks is computationally demanding. It is usually at least **PSPACE**-hard [2]. As a consequence, it is not reasonable to expect that a reduced state space method always runs quickly and produces good reduction results. It is more reasonable to expect that there is a range of methods, some of which yield better reduction results but are also more complicated and perhaps also consume more time than the others. This is precisely the situation with stubborn set and similar methods.

It is important to understand that stubborn set and similar methods are based on approximating in a safe direction. That is, they use concepts whose precise values are too expensive to find out, but for which approximations that preserve correctness are available. This is one way of coping with the high computational complexity. However, the approximations may reduce the amount of reduction obtained from the methods. This is a major, but not the only, issue in which the methods in the literature differ from each other: some use more time to compute more precise approximations than the others.

For instance, sometimes it is easy to see that a disabled Petri net transition can occur in the future; sometimes it is easy to see that it cannot; and sometimes its future is not easy to see. If we see that it cannot occur in the future, we exploit that fact, but in the two remaining cases we write “as far as we know, it may occur in the future”, and continue accordingly. Please keep in mind that when we write “as far as we know” in the sequel, it is not a vague statement but indicates the direction of approximation.

Section 2 introduces concurrent systems and their state spaces. The next two sections present so-called dynamic and static deadlock-preserving stubborn set methods using dependency graphs that have both and- and or-type of vertices. These are mostly known results, but they are presented in a general framework that is suitable for the rest of this article. Although we use stubborn set terminology, most similar methods can be understood in terms of dependency graphs. (So-called weak stubborn sets seem to need something more general than the dependency graphs of Section 4. The stubborn sets in this article are “strong”).

Section 5 investigates variability first in the choice and then in the definition of the so-called dependency relation. It is argued that there is no natural “most general” notion of dependency. The next section focuses on an issue that has been given little attention in the literature, namely the control of disabled transitions in the stubborn or similar set. Section 7 briefly discusses algorithms for actually constructing stubborn or similar sets.

Section 8 focuses on whether a stubborn set can be “best” in some non-trivial practical sense. We argue that it is not practical to define “best” as “yields smallest possible deadlock-preserving state spaces”, but it is practical, reasonable and non-trivial to define it as “in each state, expands as few transitions as the information provided by the dependency graph allows”. This means that better reduction may be possible, but not without using some idea that goes beyond dependency graphs. Of the techniques presented in the literature on stubborn sets and similar methods, most can be understood in terms of dependency graphs. As a consequence, new ideas are needed to defeat this notion of optimality. Then, using model-theoretic reasoning, a theorem is proven that states that a certain method is optimal in this sense.

## 2 Petri Nets, Concurrent Systems, and State Spaces

In this section we introduce the concurrency-related formalisms needed in this article. Let us start by specifying the kind of Petri nets we use, that is, place/transition nets, together with some related concepts.

**Definition 1.** A Petri net (or place/transition net) is a tuple  $(P, T, W, M_1)$  where  $P \cap T = \emptyset$ ,  $W$  is a function from  $(P \times T) \cup (T \times P)$  to  $\mathbb{N}$ , and  $M_1$  is a function from  $P$  to  $\mathbb{N}$ . The elements of  $P$ ,  $T$ , and  $W$  are places, transitions, and weights, respectively.  $M_1$  is the initial marking.

A marking is a function from  $P$  to  $\mathbb{N}$ .

An arc is a pair  $(x, y)$  where  $x \in P \wedge y \in T$  or  $x \in T \wedge y \in P$  such that  $W(x, y) > 0$ . If  $x \in P \cup T$ , then  $\bullet x = \{y \in T \cup P \mid W(y, x) > 0\}$  and  $x \bullet = \{y \in T \cup P \mid W(x, y) > 0\}$ .

Transition  $t \in T$  is enabled in marking  $M$ , denoted by  $M[t]$ , if and only if  $\forall p \in P : M(p) \geq W(p, t)$ . If  $t$  is enabled in  $M$ , then  $t$  may occur yielding the marking  $M'$  where  $\forall p \in P : M'(p) = M(p) - W(p, t) + W(t, p)$ . This is denoted with  $M[t] M'$ .

$M[t_1 t_2 \cdots t_n] M'$  means that there are  $M_0, \dots, M_n$  so that  $M_0 = M$ ,  $M_n = M'$ , and  $M_{i-1}[t_i] M_i$  for  $1 \leq i \leq n$ .  $M[t_1 t_2 \cdots t_n]$  means that there is an  $M'$  such that  $M[t_1 t_2 \cdots t_n] M'$ .

$M'$  is reachable from  $M$ , if and only if there are  $t_1, t_2, \dots, t_n$  such that  $M[t_1 t_2 \cdots t_n] M'$ . The set of all such  $M'$  is denoted by  $[M]$ .  $M$  is reachable if and only if  $M \in [M_1]$ .

Petri nets are an example of *concurrent systems*. We will not define concurrent systems formally. Instead, we will rely on an analogy with Petri nets, introduce some terminology, and define two kinds of state spaces formally.

A concurrent system has a set  $S_S$  of *syntactic states*. Intuitively, it is the set of all possible combinations of values of the elements that store information, such as program counters, local and global variables, and buffered communication channels. In the case of Petri nets,  $S_S$  consists of all functions from  $P$  to  $\mathbb{N}$ . The set of syntactic states is a superset of all other sets of states that we will discuss. If we say “state” without specifying its kind, then it is a syntactic state.

A concurrent system also has a set  $S_I$  of *initial states*. They are the possible syntactic states of the system when it is started. With Petri nets it is customary to specify precisely one initial state, the initial marking. However, multiple initial states are sometimes needed to model concurrent systems appropriately. This is the case, for instance, with uninitialised program variables. Multiple initial states are common in the context of temporal logics.

The set of *reachable states*  $S_A$  consists of those syntactic states that can be obtained by running the system starting at some initial state. In the case of Petri nets it is the reachable markings. Obviously  $S_I \subseteq S_A \subseteq S_S$ . We chose the subscript “A” instead of “R”, because we will later need “R” for “reduced”.

The term “syntactic states” is much less widely used than the other two. Syntactic states that are not reachable are irrelevant for the behaviour of a system. However, often it is only known after constructing the reduced state space whether some state is reachable or not. As a consequence, when checking whether two transitions depend on each other, one cannot in practice directly appeal to reachable states but must use some superset. This is why we need to discuss syntactic states.

Having introduced three sets of states, we will introduce two sets of transitions: *structural* and *semantic*. Semantic transitions are changes of the state of the system, and structural transitions are structural entities whose occurrences cause those changes. Transitions of a Petri net are an example of structural transitions. Another example is atomic statements in a concurrent programming language. Semantic transitions are triples of the form  $(s, t, s')$ , where  $s$  and  $s'$  are states,  $t$  is a structural transition, and  $s[t]s'$ .

The notation  $s[t]s'$  is extended to  $s[t_1 \cdots t_n]s'$ ,  $s[t_1 \cdots t_n]$ , and  $[s]$  like with Petri nets, and similarly with the terms “enabled” and “occur”.

We assume in this article that structural transitions are *deterministic*. That is, if  $s[t]s_1$  and  $s[t]s_2$ , then  $s_1 = s_2$ . It is common in concurrency formalisms that structural transitions are deterministic. For instance, Petri net transitions are deterministic. However, in process algebras, if actions are interpreted as structural transitions (as is reasonable), then structural transitions are not necessarily deterministic.

The following definition summarises the above notions.

**Definition 2.** A syntactic state space is a triple  $(S_S, T, \delta_S)$ , where  $\delta_S$  is a partial function from  $S_S \times T$  to  $S_S$ . The elements of  $S_S$ ,  $T$ , and  $\delta_S$  are syntactic states, structural transitions, and semantic transitions, respectively. By  $s[t]$  we mean that  $\delta_S(s, t)$  is defined, and by  $s[t]s'$  we mean that  $\delta_S(s, t) = s'$ .

Let  $(S_S, T, \delta_S)$  be a syntactic state space and  $S_I \subseteq S_S$ . The tuple  $(S_A, T, \delta_A, S_I)$  is a full state space if and only if  $S_A$  and  $\delta_A$  are the smallest subsets of  $S_S$  and  $\delta_S$  that satisfy

- $S_I \subseteq S_A$ , and
- if  $s \in S_A$  and  $s[t]s'$ , then  $s' \in S_A$  and  $(s, t, s') \in \delta_A$ .

The elements of  $S_I$ ,  $S_A$ , and  $\delta_A$  are initial states, reachable states, and reachable transitions, respectively.

### 3 Reduced State Spaces That Preserve Deadlocks

In this section we develop the theory of reduced state spaces up to a theorem that promises that deadlocks are preserved if certain conditions are met. The conditions are *dynamic*, meaning that they refer to the future states of the current state, and thus cannot be easily evaluated in the current state. Therefore, they do not immediately yield a method for constructing reduced state spaces. However, they are an intermediate step towards such a method. Sets that satisfy the conditions are called *deadlock-preserving strong dynamic stubborn sets* in [11], and *strongly dynamically stubborn sets* in [12].

The construction of the state space of a concurrent system maintains a set of found states. Initially the initial states are found. Each found state  $s$  is *expanded* by finding all structural transitions  $t$  and states  $s'$  such that  $s[t]s'$ . If  $s'$  has not yet been found, it is marked as found and will be expanded later. In general, unexpanded found states may be picked for expansion in any order, and the expansion of a state may interleave with the expansion of another state.

In the stubborn set and similar methods, not necessarily all enabled structural transitions are used when expanding a state. This motivates the notion of *reduced state space*, defined next.

**Definition 3.** Let  $(S_S, T, \delta_S)$  be a syntactic state space. A reduced state space generator is a function from  $S_S$  to  $2^T$ .

Let  $R$  be a reduced state space generator and  $S_I \subseteq S_S$ . The reduced state space generated by  $R$  is the tuple  $(S_R, T, \delta_R, S_I)$ , where  $S_R$  and  $\delta_R$  are the smallest subsets of  $S_S$  and  $\delta_S$  that satisfy

- $S_I \subseteq S_R$ , and
- if  $s \in S_R$ ,  $t \in R(s)$ , and  $s[t]s'$ , then  $s' \in S_R$  and  $(s, t, s') \in \delta_R$ .

Let  $en(s) = \{t \mid s[t]\}$ . If  $R_1$  and  $R_2$  are two reduced state space generators such that  $R_1(s) \cap en(s) \subseteq R_2(s) \cap en(s)$  for every  $s \in S_S$ , then  $R_1$  obviously generates a smaller or the same state space as  $R_2$  for the same set of initial states.

In this article we are interested in reduced state spaces that have precisely the same deadlocks as the corresponding full state spaces. Furthermore, a state that looks like a deadlock in a reduced state space must indeed be a deadlock.

**Definition 4.** Let  $(S_S, T, \delta_S)$  be a syntactic state space. A deadlock is an  $s \in S_S$  such that  $s[t]$  holds for no  $t \in T$ .

Let  $(S_A, T, \delta_A, S_I)$  be a full and  $(S_R, T, \delta_R, S_I)$  a reduced state space. The latter preserves deadlocks if and only if

- every deadlock that is in  $S_A$  is in  $S_R$ , and
- if  $s \in S_R$  and  $s$  is not a deadlock, then there is  $t \in R(s)$  such that  $s[t]$ .

**Theorem 1.** Let  $(S_S, T, \delta_S)$  be a syntactic state space. Assume that  $R$  satisfies the following for every  $s \in S_S$ ,  $t \in R(s)$ ,  $t_1 \in T \setminus R(s)$ ,  $\dots$ ,  $t_n \in T \setminus R(s)$ .



- D0** If  $s$  is not a deadlock, then there is  $t' \in R(s)$  such that  $s[t']$ .  
**D1** If  $s[t_1 \cdots t_n t] s'$ , then  $s[tt_1 \cdots t_n] s'$ .  
**D2** If  $s[t]$  and  $s[t_1 \cdots t_n] s'$ , then  $s'[t]$ .

Then the reduced state spaces generated by  $R$  preserve deadlocks.

*Proof.* We prove first by induction that if  $s[t_1 \cdots t_n] s'$ ,  $s \in S_R$ , and  $s'$  is a deadlock, then  $s' \in S_R$ . This is obvious when  $n = 0$ . If  $n > 0$ , then  $s[t_1]$ . This implies by **D0** that there is  $t \in R(s)$  such that  $s[t]$ . If none of  $t_1, \dots, t_n$  is in  $R(s)$ , then **D2** yields  $s'[t]$ , which contradicts the assumption that  $s'$  is a deadlock. Therefore, there is  $1 \leq i \leq n$  such that  $t_i \in R(s)$ . By choosing the smallest such  $i$  we get  $t_1 \notin R(s), \dots, t_{i-1} \notin R(s)$ . Now **D1** yields an  $s''$  such that  $s[t_i] s'' [t_1 \cdots t_{i-1} t_{i+1} \cdots t_n] s'$ . We have  $s'' \in S_R$ , from which the induction assumption gives  $s' \in S_R$ .

If  $s'$  is a deadlock in the full state space, then there are  $s \in S_I$  and  $t_1 \in T, \dots, t_n \in T$  such that  $s[t_1 \cdots t_n] s'$ . By Definition 3  $s \in S_R$ , so by the above result  $s' \in S_R$ . On the other hand, if  $s \in S_R$  and  $s$  is not a deadlock, then **D0** implies that  $s[t]$  for some  $t \in R(s)$ .  $\square$

Theorem 1 is from 9. As was mentioned towards the beginning of this section, sets that satisfy **D0**, **D1**, and **D2** are called deadlock-preserving strong dynamic stubborn sets in 11. “Strong” refers to the fact that there is also a *weak* stubborn set theory where **D0** and **D2** have been replaced by a weaker requirement 10,9. Weak stubborn sets provide better reduction results than strong stubborn sets. However, they are more difficult to construct and have found little use. Therefore, we ignore them in this article. The weak version of stubborn sets is explored in 12, in which they are called simply *dynamically stubborn sets*.

As was mentioned above, “dynamic” refers to the fact that the conditions talk about states that are in the future of  $s$ , and cannot thus be easily evaluated if only the current state is known. In the next section we will introduce “static” conditions whose evaluation only needs the current state. The use of the words “static” and “dynamic” resembles their use in the theory of programming languages.

The word “stubborn” reflects the intuition that the stubborn set does not let the outside world affect what it will do. Letting  $t_1, \dots, t_n$  occur first does not enable or disable  $t$ , and the state after they all have occurred does not depend on whether  $t$  occurred first or last.

## 4 Dependency Graphs and Static Stubborn Sets

In this section we aim at *static* concepts of stubborn sets, that is, such concepts that the set can be constructed on the basis of the current state, without knowing its future states. This is important, because the future states are not known when the state is expanded.

Many concepts of that kind have been presented in the literature. Most of them use some kind of *dependency relation*, defined next.

**Definition 5.** Let  $(S_S, T, \delta_S)$  be a syntactic state space and  $S \subseteq S_S$ . A dependency relation with respect to  $S$  is any symmetric binary relation  $\mathcal{D} \subseteq T \times T$  such that for every  $t_1 \in T$ ,  $t_2 \in T$ , and  $s \in S$ , if  $(t_1, t_2) \notin \mathcal{D}$ ,  $s[t_1]$ , and  $s[t_2]$ , then there is some  $s' \in S_S$  such that  $s[t_1 t_2] s'$  and  $s[t_2 t_1] s'$ .

The definition differs from the commonly presented in that it has the  $S$  to which the “commutativity requirement” is restricted. In the literature, commutativity is usually required for “all states”, meaning either that  $S = S_S$  or  $S = S_A$ . We will also discuss other choices of  $S$  in Section 5. Another difference is that we do not require that the relation is reflexive. Whether or not it is reflexive is irrelevant in this article.

It is natural to require that the relation is symmetric, because the commutativity requirement in it is symmetric. Even if the symmetry requirement were removed, either both  $(t_1, t_2) \in \mathcal{D}$  and  $(t_2, t_1) \in \mathcal{D}$  would have to hold to satisfy the definition, or it would not matter whether both or one or the other or neither holds.

Please notice that  $\mathcal{D}$  need not be the smallest relation that has the properties mentioned in the definition. This is an example of approximating in the safe direction. It would be correct to let  $\mathcal{D}$  consist of all pairs of structural transitions, but then the method would not give any reduction of the state space.

The dependency relation is used in various ways in the literature, and there is also another important issue: the controlling of disabled structural transitions. To discuss a wide range of possibilities, we introduce an abstract notion of *dependency graphs*. The intuition underlying the definition is explained immediately after it.

**Definition 6.** A dependency graph  $(E, D, C, “\rightsquigarrow”)$  is a directed graph that has three kinds of vertices, enabled ( $E$ ), disabled ( $D$ ), and condition ( $C$ ), and satisfies the following.

1.  $E \cap D = E \cap C = D \cap C = \emptyset$ .
2. “ $\rightsquigarrow$ ”  $\subseteq ((E \cup C) \times (E \cup D)) \cup (D \times C)$ . By  $v \rightsquigarrow v'$  we mean that  $(v, v') \in “\rightsquigarrow”$ .
3. If  $v \in E$ , then  $v \not\rightsquigarrow v$ . (Part 2 implies  $v \not\rightsquigarrow v$  when  $v \in D \cup C$ .)
4. If  $v_1 \in E$ ,  $v_2 \in E$ , and  $v_1 \rightsquigarrow v_2$ , then  $v_2 \rightsquigarrow v_1$ .
5. If  $v_1 \in D$ , then there is at least one  $v_2$  such that  $v_1 \rightsquigarrow v_2$ .

Let  $(E, D, C, “\rightsquigarrow”)$  be a dependency graph. If  $V_1 \subseteq E \cup D \cup C$  and  $V_2 \subseteq E \cup D \cup C$ , then  $V_1 \sqsubseteq V_2$  denotes that  $V_1 \cap E \subseteq V_2 \cap E$ . If  $v \in E \cup D \cup C$ , then  $\bullet v = \{u \mid u \rightsquigarrow v\}$  and  $v \bullet = \{u \mid v \rightsquigarrow u\}$ .

The idea is that enabled and disabled vertices correspond to enabled and disabled structural transitions of the concurrent system. Condition vertices correspond to reasons why a structural transition is disabled. An empty input place of a Petri net transition is a good example of such a reason. The edge  $d \rightsquigarrow c$  where  $d \in D$  and  $c \in C$  represents the situation that  $d$  is disabled because the condition represented by  $c$  does not hold in the current state. There may be many such  $c$  for each  $d$ . For instance, a Petri net transition may have many empty input

places. The edge  $c \rightsquigarrow t$  models the fact that, as far as we know, an occurrence of the structural transition  $t$  may make  $c$  hold. We will make these and the succeeding ideas precise in Definition 7.

Part 2 of the definition stipulates that all output edges of disabled vertices lead to condition vertices and all input edges of condition vertices come from disabled vertices. This is in harmony with the role of condition vertices.

Definition 6 reflects the thinking that the enabling condition of a structural transition is a conjunction of individual conditions. With Petri nets this is naturally the case. With other formalisms, if we do not know how to divide the enabling condition to more than one conjunct, we can let  $d$  have precisely one output vertex, from which there is an edge to each structural transition that, as far as we know, can enable  $d$ . Part 5 requires that each disabled structural transition has at least one conjunct. It simplifies part 3 of Definition 8 that will be presented later. It does not imply loss of generality, because, if necessary, one may add an extra  $c$  that does not have output edges. Such a  $c$  represents the conjunct “False”.

An edge  $e \rightsquigarrow t$  where  $e$  is an enabled vertex and  $t \neq e$  models the fact that the occurrence of transition  $t$  may, roughly speaking, modify the effect of  $e$  or disable  $e$ . More precisely, there is some dependency relation  $\mathcal{D}$ , and the edge is drawn if and only if  $(e, t) \in \mathcal{D}$ . (Dependency graphs will be used in such a way that the current state is in the  $S$  of the dependency relation.) Dependency relations are symmetric. Therefore, if the edge  $e \rightsquigarrow t$  is drawn and also  $t$  is enabled, then also the edge  $t \rightsquigarrow e$  must be drawn. This motivates part 4 of Definition 6.

Part 3 is because, as we will see in Definition 8, the edge  $e \rightsquigarrow e$  where  $e \in E$  would represent a requirement of the form “if  $e$  is in  $V$  then  $e$  must be in  $V$ ”. The requirement obviously holds automatically and thus need not be stated.

The following definition makes it precise what it means that a dependency graph is “correct” for a concurrent system.

**Definition 7.** A dependency graph  $(E, D, C, “\rightsquigarrow”)$  respects the syntactic state space  $(S_S, T, \delta_S)$  in  $s \in S_S$  if and only if the following hold:

1.  $E = \{t \in T \mid s[t]\}$
2.  $D = T \setminus E$
3. If  $d \in D$  and  $s[t_1 t_2 \cdots t_n d]$ , then, for each  $c$  such that  $d \rightsquigarrow c$ , there is  $1 \leq i \leq n$  such that  $c \rightsquigarrow t_i$ .
4. There is a dependency relation  $\mathcal{D}$  with respect to the states that are reachable from  $s$  such that if  $e \in E$ ,  $t \neq e$ , and  $(e, t) \in \mathcal{D}$ , then  $e \rightsquigarrow t$ .

The goal of the dependency graph is to make it possible to compute a set that does not necessarily contain all enabled structural transitions, but, even so, to find deadlocks, it suffices to let the enabled structural transitions in the set occur. The next definition characterises such a set, and the theorem after the definition states that it indeed does so.

**Definition 8.** A stubborn set of a dependency graph is a collection  $V$  of vertices such that the following hold:

1. If  $E \neq \emptyset$  then  $V \cap E \neq \emptyset$ .
2. If  $v \in V \setminus D$  then  $v \bullet \subseteq V$ .
3. If  $v \in V \cap D$  then  $v \bullet \cap V \neq \emptyset$ .

That is, it contains all output vertices of its enabled and condition vertices, and at least one output vertex of each of its disabled vertices. It also contains at least one enabled vertex, if there are any.

**Theorem 2.** *Let  $(S_S, T, \delta_S)$  be a syntactic state space. For each  $s \in S_S$ , let  $V(s)$  be a stubborn set of a dependency graph that respects  $(S_S, T, \delta_S)$  in  $s$ , and let  $R(s) = V(s) \cap T$ . Then  $R$  has the properties **D0**, **D1**, and **D2** in Theorem 1, and the reduced state spaces generated by  $R$  preserve deadlocks.*

*Proof.* By Definitions 7(1) and 8(1), if  $s$  is not a deadlock, then  $E(s)$ ,  $V(s) \cap E(s)$ , and  $R(s)$  contain an enabled structural transition. So **D0** holds.

During the rest of the proof, let  $s_0 = s$ ,  $t \in R(s_0)$ ,  $t_1 \notin R(s_0)$ ,  $\dots$ ,  $t_n \notin R(s_0)$ , and  $s_0 [t_1] s_1 [t_2] \dots [t_n] s_n$ .

Assume that  $s_0 [t] s'_0$ . By Definitions 7(1), 8(2), and 6(2),  $t \in E(s_0)$  and  $t \bullet \subseteq V(s_0) \cap T$ . Because  $t_i \notin R(s_0)$  for  $1 \leq i \leq n$ , we have  $t \not\rightsquigarrow t_i$ . If  $\mathcal{D}(s_0)$  is the dependency relation in Definition 7(4), then  $(t, t_i) \notin \mathcal{D}(s_0)$ . Repeated application of Definition 5 yields  $s'_1, \dots, s'_n$  such that  $s'_0 [t_1] s'_1 [t_2] \dots [t_n] s'_n$  and  $s_i [t] s'_i$  for  $1 \leq i \leq n$ . So **D2** holds.

Assume that  $s_n [t]$ . If  $t \in E(s_0)$ , then **D1** holds by what was proven above because  $t$  is deterministic. We show that the opposite case  $t \in D(s_0)$  is not possible, by assuming it and deriving a contradiction. By Definitions 8(3), 8(2), and 6(2), there is a  $c \in t \bullet$  such that  $c \in V(s_0)$  and  $c \bullet \subseteq R(s_0)$ . By Definition 7(3), there is  $1 \leq i \leq n$  such that  $t_i \in c \bullet$ . But this contradicts  $t_i \notin R(s_0)$ . This completes the proof of **D1**.  $\square$

We conclude this section by defining and commenting on three dependency graphs. We skip the (simple) proofs that they indeed are dependency graphs and respect the syntactic state space of the Petri net or concurrent program as promised.

*Example 1.* Let  $(P, T, W, M)$  be a Petri net. The following graph  $(E, D, C, \sim)$  is a dependency graph that respects  $(P, T, W, M)$ . (That is, it respects the syntactic state space of  $(P, T, W, M)$  in  $M$ .)

- $E = \{t \in T \mid M[t]\}$
- $D = T \setminus E$
- $C = \{p \in P \mid \exists t \in T : M(p) < W(p, t)\}$
- $(t_1, t_2) \in \mathcal{D}$  if and only if  $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ .
- $t \sim t'$  where  $t \in E$  and  $t' \in T$  if and only if  $t' \neq t$  and  $(t, t') \in \mathcal{D}$ .
- $t \sim p$  where  $t \in D$  and  $p \in C$  if and only if  $M(p) < W(p, t)$ .
- $p \sim t$  where  $p \in C$  and  $t \in T$  if and only if  $t \in \bullet p$ .

$\square$

*Example 2.* Let everything be like in Example 1, except the following:

- $(t_1, t_2) \in \mathcal{D}$  if and only if
 
$$\exists p \in P : \min(W(t_1, p), W(t_2, p)) < \min(W(p, t_1), W(p, t_2)).$$
- $p \rightsquigarrow t$  where  $p \in C$  and  $t \in T$  if and only if
 
$$W(p, t) < W(t, p) \wedge W(p, t) \leq M(p).$$

The resulting graph is a dependency graph that respects  $(P, T, W, M)$ .  $\square$

It is not difficult to check that the edges of the graph in Example 2 are a subset of the edges of the graph in Example 1. This implies that all stubborn sets of Graph 1 are stubborn sets of Graph 2, but not necessarily vice versa. Graph 2 may have stubborn sets that yield smaller  $R(s) \cap E(s)$  than any stubborn set of Graph 1. Therefore, Graph 2 has the potential of yielding better reduction results than Graph 1. On the other hand, Graph 1 has a simpler definition. This is an example of the trade-off between simplicity and reduction power mentioned in the introduction.

*Example 3.* Consider a concurrent program consisting of sequential processes and shared variables. Each sequential process has a program counter  $pc_i$ . Each atomic statement of process  $i$  has an enabling condition of the form  $pc_i = c \wedge \varphi$ , where  $c$  is a constant and  $\varphi$  is a Boolean expression on values of shared variables. If a shared variable occurs in  $\varphi$ , we say that the statement tests the variable. When executed, a statement may read and write values of shared variables. It also may (and usually does) modify the value of the program counter. It is assumed that atomic statements are deterministic.

To construct a dependency graph for a program in a given state, let  $E$  and  $D$  be the enabled and disabled atomic statements, respectively. Let  $C$  consist of the program counters together with one condition  $c_t$  for each atomic statement  $t$ . Every atomic statement  $t$  whose  $pc_i = c$  does not hold has  $t \rightsquigarrow pc_i$ , and  $pc_i \rightsquigarrow t'$  for every atomic statement  $t'$  of process  $i$ . Every atomic statement  $t$  whose  $\varphi$  does not hold has  $t \rightsquigarrow c_t$ , and  $c_t \rightsquigarrow t'$  for every atomic statement  $t'$  that writes to any shared variable that  $t$  tests. Two atomic statements depend on each other if and only if they belong to the same process and expect its program counter have the same value, or they belong to different processes and one of them writes to a shared variable that the other tests, reads, or writes.  $\square$

Example 3 uses coarse approximations. A better, but more complicated, dependency graph may be obtained by dividing each  $\varphi$  to more than one conjunct where possible, and by analysing more carefully whether a transition can really make a condition hold or interfere with the execution of another transition. We will discuss the latter in Example 4.

All three examples have the important property that to construct a dependency graph, no information on other states than the current state is needed. This implies that a stubborn set can be computed and the state can be expanded without knowledge of future states, as is necessary when constructing a reduced state space in practice. Algorithms for computing a stubborn set given a dependency graph are briefly discussed in Section 7.

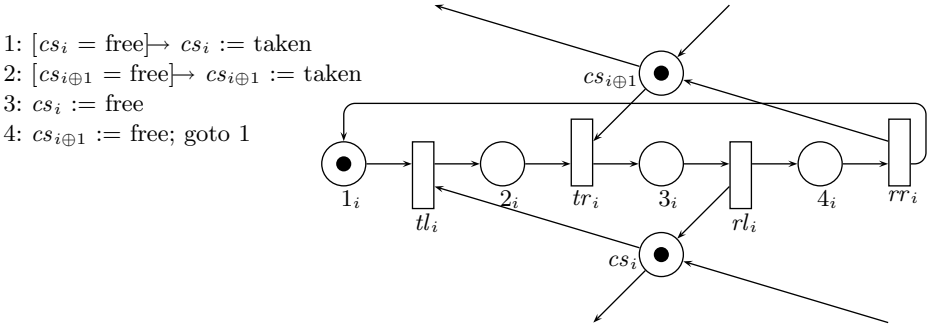


Fig. 1. One dining philosopher as program code and Petri net

## 5 The Effect of Dependency Relations

In this section we first present an example that demonstrates that the replacement of a dependency relation by a more precise one may have a dramatic effect on reduction results. Furthermore, the way in which the system is modelled determines how easy it is to find the better dependency relation. Then we discuss options for the set with respect to which dependency relations are defined.

*Example 4.* The well-known *dining philosophers' system* consists of  $n$  philosophers and  $n$  chop sticks. Figure 1 shows one philosopher in two formalisms. In the figure,  $i \oplus 1$  denotes  $(i \bmod n) + 1$ ,  $cs$  refers to a chop stick,  $tl$  abbreviates “take left”,  $rr$  is “return right”, and  $tr$  and  $rl$  follow the same logic. Although the last four abbreviations are only shown in the Petri net model, they can also be interpreted as statement names in the other model. To simplify the discussion we assume that  $n \geq 2$ . Let  $1 \ominus 1 = n$  and  $i \ominus 1 = i - 1$  if  $2 \leq i \leq n$ .

Let  $x_i$  denote  $2_i$  or  $pc_i$ . We denote the initial state with  $1^n$ . In it,  $tl_{i \oplus 1} \rightsquigarrow tr_i \rightsquigarrow x_i \rightsquigarrow tl_i$  in every correct dependency graph, implying that all enabled transitions must be investigated. In the next state, one philosopher is in state 2. To avoid notational problems, we assume that it is philosopher 1; the other cases are symmetric. The state is thus  $21^{n-1}$ . In this and all other states of the form  $2^i 1^{n-i}$ , where  $0 < i < n$ ,  $\{tr_i, tl_{i \oplus 1}\}$  is a stubborn set, and it is the best possible. There are algorithms that find it. If  $tl_{i \oplus 1}$  occurs, the resulting state is of the same form or it is the deadlock state  $2^n$ . If  $tr_i$  occurs, the state  $2^{i-1} 31^{n-i}$  is obtained. Here things become different depending on the dependency relation.

With the Petri net model and the dependency relation of Example 1 or 2,  $\{rl_i\}$  is stubborn. Typical algorithms find it and construct the state  $2^{i-1} 41^{n-i}$ . There  $\{rr_i\}$  is stubborn and found by typical algorithms, yielding  $2^{i-1} 1^{n-i+1}$ . It is of a form that we have already analysed. Altogether  $3n^2 - 3n + 2$  states are constructed [10].

On the other hand, with the concurrent program model and the dependency relation in Example 3,  $rl_i$ ,  $tr_{i \oplus 1}$ , and  $rr_{i \oplus 1}$  write to the same shared variable. As a consequence,  $rl_i \rightsquigarrow tr_{i \oplus 1}$  and  $rl_i \rightsquigarrow rr_{i \oplus 1}$  in  $2^{i-1} 31^{n-i}$ . This does not

happen with Petri nets, because introducing more tokens to a place cannot disable transitions, and if two transitions occur, the joint effect is independent of the order. This does happen with concurrent programs, because, in general, a write access to a shared variable may disable another statement that accesses the variable, and the joint effect of two write accesses may depend on their order.

The precise effect of these extra “ $\rightsquigarrow$ ”-edges depends on the stubborn set construction algorithm. With typical algorithms, all enabled transitions are investigated in all states where each philosopher is in state 1 or 3, because then each philosopher “pays attention” to her or his left. The state  $2^{i-1}31^{n-i}$  is such when  $i = 1$ . This leads to the construction of many states, including, if  $n$  is even, all that are of the form  $x_1x_21 \cdots x_{n/2}1$ , where each  $x_i$  is either 1 or 3. There are  $2^{n/2}$  such states. The reduced state space has grown from quadratic to exponential.

This growth could be avoided by doing more precise dependency analysis. That could be made relatively easy by modelling the system in a different way. Reading from a bounded or unbounded fifo queue can be considered independent of writing to it, so one possibility is to model the chop sticks with fifo queues. This solves  $rl_i \rightsquigarrow tr_{i \oplus 1}$ , but not  $rl_i \rightsquigarrow rr_{i \oplus 1}$ . A complete solution is to model the states of the chop sticks with three values: free, taken\_left, and taken\_right; and to add the guards “[ $cs_i = \text{taken\_left}$ ]→” and “[ $cs_{i \oplus 1} = \text{taken\_right}$ ]→” to lines 3 and 4, respectively. Then  $rl_i$ ,  $tr_{i \oplus 1}$ , and  $rr_{i \oplus 1}$  are never simultaneously enabled, and can thus be considered independent of each other.  $\square$

Definition 5 has a set  $S$ , with respect to which the dependency relation is defined. Until now we have had  $S = S_S$  in our examples. However, Definition 7 only requires that  $S$  contains the states that are reachable from the current state. This can be exploited when the program code of a process does not consist of one big loop. When the process has executed or bypassed a statement to which it can never return, the statement can be declared independent of all statements, because from then on it is never enabled. This kind of an idea was applied to Büchi automata type of verification in 8.

If we look carefully at the proof of Theorem 2, it is not even necessary that  $S$  covers all states that are reachable from the current state. It suffices that it covers those states which may be reached without occurrences of transitions from the stubborn set. Therefore,  $S$  could be made smaller, potentially reducing the number of “ $\rightsquigarrow$ ”-edges and improving reduction results.

Unfortunately, this idea has a drawback as we show next. It is obvious from Definition 5 that for each fixed  $S$ , there is a unique “most precise” dependency relation that only spans the “ $\rightsquigarrow$ ”-edges that are absolutely necessary. As we will show, when  $S$  is made smaller than in Definition 7, it is not any more fixed and uniqueness is lost. This makes it complicated to fully exploit this idea.

The Petri net in Figure 2 does not have disabled transitions, so its dependency graph and stubborn sets are determined solely by the dependency relation. Clearly  $\{c\}$  and  $\{a, c\}$  are not its stubborn sets, because  $bb$  disables  $c$ , violating **D2**. Neither is  $\{a\}$ , because  $s[bca]$  but  $\neg(s[abc])$ , violating **D1**. That  $s[acb]$  but  $\neg(s[bac])$  rules out  $\{b\}$ . However,  $\{b, c\}$  satisfies **D2** and **D1**: the only outside transition is  $a$ , it cannot occur twice, it does not disable  $b$  or  $c$ , and there



Fig. 2. An example Petri net and its state space

are states  $s_{ab}$  and  $s_{ac}$  such that  $s[ab] s_{ab}$ ,  $s[ba] s_{ab}$ ,  $s[ac] s_{ac}$ , and  $s[ca] s_{ac}$ . A similar analysis shows that also  $\{a, b\}$  satisfies **D2** and **D1**. There is no unique minimum stubborn set, so there cannot be any unique minimum dependency relation either.

If  $S$  is not liberated beyond Definition 7, then  $c$  depends both on  $a$  and on  $b$ , because after  $b$ ,  $a$  and  $b$  disable  $c$ . Then the only stubborn set is  $\{a, b, c\}$ .

In conclusion, liberating  $S$  can improve reduction results. However, the relation that justifies the improved results is sensitive to what other transitions are in the stubborn set. It is perhaps better not to call such a relation “dependency”.

## 6 Controlling Disabled Transitions

In this section we discuss conditions with which part 3 of Definition 7 is established. This topic has not received enough attention in the literature. Many algorithms, including the ample set algorithm of [1], “algorithm 2” in [6], and even [3], do not utilise the information needed to handle disabled transitions and their enabling conditions. Many other publications address the topic as a side issue that is not made particularly explicit. We are not aware of any research that measures reduction gains from using finer-grained information about enabling conditions, aside from [4].

The first remark has a message similar to Example 4: more careful analysis of which structural transitions can make a condition hold yields better reduction results. Consider a structural transition that is disabled because a shared variable  $v$  has a wrong value. In the absence of additional information, every write access to  $v$  potentially enables the transition. However, if the test is of the form “ $v = 3$ ”, then only those writing accesses whose result may be 3 need to be taken into account. In Section 4 we discussed the division of the enabling condition into more than one conjunct, which also relies on analysing the semantics of individual statements more carefully.

Some stubborn set and similar methods base the controlling of disabled transitions on the assumption that the concurrent system is a collection of sequential processes. They find a subset of processes and use their enabled transitions for expanding the current state. They rely on the notion of “next” transitions of a process, that is, those whose test on the program counter evaluates to **True**. For each process  $i$  that is in the set, also those processes must be in the set that may enable disabled next transitions of process  $i$  or that have transitions that are dependent with enabled next transitions of process  $i$ .



Such methods can use dependency graphs whose vertices are processes and conditions instead of (enabled and disabled) transitions and conditions. However, they can also be understood in terms of the dependency graphs of this article.

The “next” transitions of a process usually depend on each other, because the execution of any of them changes the value of the program counter and thus disables the others. Even when this is not precisely true, assuming that they all depend on each other is an approximation in the safe direction. The structural transitions that are not “next” are disabled by the program counter. So they can be controlled by taking all structural transitions that write to the program counter – that is, all structural transitions of the process – into the stubborn set. Therefore, it is correct, although not necessarily optimal regarding reduction results, to use processes instead of individual transitions as the basic units of stubborn set construction.

Using processes instead of structural transitions is faster, because computation proceeds in bigger steps. In particular, with a typical Petri net model using individual transitions, the analysis may follow long chains backwards the control structure of the process until it finds the place where the control token is, consuming time. An example of such a chain is  $rr_i \rightsquigarrow 4_i \rightsquigarrow rl_i \rightsquigarrow 3_i \rightsquigarrow tr_i \rightsquigarrow 2_i \rightsquigarrow tl_i \rightsquigarrow 1_i$  in Figure 1.

On the other hand, the use of processes as basic units also has drawbacks. It introduces a “ $\rightsquigarrow$ ”-path to the enabled transitions of the process even if there is no path from them to the disabled transition in question in the control structure of the process. The use of transitions as basic units avoids that, because then the “ $\rightsquigarrow$ ”-paths need not exit the disabled transitions of the process. The same benefit could be obtained by somehow detecting that the disabled transition in question cannot be enabled any more, so it can be declared independent of all other transitions according to Definition 5.

The use of individual transitions also has the advantage that “ $\rightsquigarrow$ ”-paths along disabled transitions may exit the process. If a structural transition is disabled both because of the program counter and because of a shared variable, the path may choose either one. Consider Figure 1 modified so that  $cs_{i\oplus 1}$  is empty. Then  $tl_{i\oplus 1} \rightsquigarrow cs_{i\oplus 1} \rightsquigarrow rr_i \rightsquigarrow 4_i \rightsquigarrow rl_i \rightsquigarrow 3_i \rightsquigarrow tr_i$ . Because both input places of  $tr_i$  are empty, we may choose to continue either with  $cs_{i\oplus 1}$  or with  $2_i$ . If we choose  $2_i$  as methods using processes as basic units would implicitly do, then we would have to take the enabled transition  $tl_i$  and continue to  $tr_{i\ominus 1}$ . However, if we choose  $cs_{i\oplus 1}$ , then we are back where we were a moment ago and need not investigate more transitions and places. We had to take the transitions  $rr_i$ ,  $rl_i$ , and  $tr_i$ , but it does not matter, because they are all disabled.

More options imply potentially more stubborn sets and better chances that there is a good one among them. However, it may be difficult to exploit the possibilities. Part 2 of Definition 8 is easy to use: if  $v$  is considered, then every  $v' \in v\bullet$  must be considered. On the other hand, part 3 forces to choose one element from  $v\bullet$  for consideration. It may be difficult to find out which element is the best choice. Fixing the choice in advance, like methods using processes as basic units do, saves us from the pain of choosing, but the pre-determined

choice is not necessarily the best possible. Furthermore, when a “next” structural transition is disabled by two different shared variables, we have to choose between them even if we use processes as basic units.

It is reasonable to ask whether there are good stubborn set construction algorithms that can exploit the possibility to choose. It is the time to discuss briefly the algorithm issue.

## 7 Algorithms

Consider two stubborn sets that do not have enabled transitions in common. Little is known about how to find out which one is better. It depends not only on the behaviour of the system, but also on what stubborn sets are used in other states. If a stubborn set has ten enabled transitions and they all lead to states that have already been found via other paths, then it is better than another stubborn set that has one enabled transition that leads to a new state. This is a potentially good topic for future research.

On the other hand, it is obvious that if  $V_1 \sqsubset V_2$ , then  $V_1$  is better, where “ $\sqsubset$ ” was defined in Definition 6. (The situation may be different with methods that preserve properties other than deadlocks, because there an extra transition in one state may save many transitions in other states.) We will concentrate on this guideline in the rest of this article.

If the choices involved in part 3 of Definition 8 are fixed, then the problem is theoretically simple: compute a “ $\rightsquigarrow$ ”-closed set of vertices such that an enabled vertex is included. “ $\sqsubset$ ”-minimal closed sets can be found in linear time with strong component algorithms (see, e.g., [11]). However, often it is possible to find a small stubborn set faster by computing a closed set starting with an enabled transition until either it is complete or computation becomes “too difficult”. In the latter case, the set of all enabled transitions is used. Let us call such methods *fallback methods*. “Too difficult” may mean, for instance, that computation enters a disabled transition in a process that has not yet been considered.

The next example demonstrates that fallback methods run a risk.

*Example 5.* Consider a banquet where there are  $m$  tables, each with  $n > 2$  dining philosophers like in Example 4. Dependency analysis is assumed precise enough to yield  $3n^2 - 3n + 2$  states when  $m = 1$ . Assume that the tables are ordered, and the first table that is not deadlocked is used as the starting point of the computation of stubborn sets. Then the above-mentioned strong component algorithms investigate one table at a time, the previous tables being deadlocked and the succeeding tables in their initial states. This yields  $m(3n^2 - 3n + 1) + 1$  states.

Now consider a fallback method that tries each process in turn until it finds a stubborn set that consists of at most two processes, or runs out of processes. It yields otherwise the same results as the previous method but, when every table is deadlocked or in its initial state, it reverts to the fallback and uses every table that is not deadlocked. It is still the case that at most one table can be in a state other than the initial or deadlock state, but now any table can be in its initial or deadlock state independently of the others. This yields  $m2^{m-1}(3n^2 - 3n) + 2^m$  states.

Both methods in this example only use information provided by the dependency graph. It is clear that the former method makes better use of that information than the latter. In the next section we prove that it is possible to *fully* exploit the information in a dependency graph.  $\square$

A quadratic algorithm is known for computing “ $\sqsubseteq$ ”-minimal stubborn sets in the presence of the choices involved in part 3 of Definition 8 (see, e.g., [11,12]). It starts with some stubborn set (such as  $EUDUC$ ) and tries to remove one enabled vertex at a time. Removal consists of traversing the “ $\rightsquigarrow$ ”-edges backwards and removing each encountered enabled and condition vertex, and each disabled vertex whose every immediate successor has been removed. If the set does not contain any enabled vertices after the removal, the removal is cancelled.

This algorithm is somewhat expensive. However, its cost is less of a problem, if it is used to fine-tune a stubborn set that has been constructed with cheaper means.

## 8 An Optimality Result

In this section we first develop a non-trivial practical notion of optimal stubborn sets, and then prove that certain stubborn sets are such.

It is obvious that to obtain maximal reduction, we should prefer stubborn sets that are minimal with respect to “ $\sqsubseteq$ ”. However, there may be sets that are smaller still with respect to “ $\sqsubseteq$ ”, not stubborn, yet still suffice to preserve deadlocks in the sense of the following definition.

**Definition 9.** *Let  $(S_S, T, \delta_S)$  be a syntactic state space and let  $s \in S_S$ . The set  $T' \subseteq T$  is deadlock-preserving in  $s$ , if and only if either  $s$  is a deadlock, or  $T'$  contains an enabled transition and, for every  $n \in \mathbb{N}$  and every deadlock  $s_d$  that is reachable from  $s$  by  $n$  transition occurrences, there are  $t \in T'$  and  $s'$  such that  $s \{t\} s'$  and  $s_d$  is reachable from  $s'$  by less than  $n$  transition occurrences.*

Stubborn sets are deadlock-preserving, but not necessarily vice versa. In Figure 2,  $\{a\}$  is deadlock-preserving but not stubborn. Unfortunately, in practice we cannot build a reduced state space generation method on finding minimal deadlock-preserving sets. The following theorem tells that they are too hard to find. A Petri net is *1-safe* if and only if each of its places never contains more than one token.

**Theorem 3.** *Let  $T'$  be a subset of the transitions of a 1-safe Petri net. The problem of testing whether  $T'$  is deadlock-preserving is **PSPACE-hard**.*

*Proof.* Testing whether a 1-safe Petri net has deadlocks is **PSPACE-hard** [2]. Assume that a 1-safe Petri net whose transitions are  $t_1, \dots, t_m$  has been given. Extend it by adding places  $\hat{p}$ ,  $p$ , and  $p'$ , transitions  $t$  and  $t'$ , and arcs  $(\hat{p}, t)$ ,  $(t, p)$ ,  $(\hat{p}, t')$ ,  $(t', p')$ , and  $(p', t_i)$  and  $(t_i, p')$  for each  $1 \leq i \leq m$ . Of the added places,  $\hat{p}$  is marked initially and the others are not. Now  $\{t\}$  is deadlock-preserving if and only if the given net does not have deadlocks.  $\square$

Minimal deadlock-preserving sets would be a very appropriate notion of optimal sets, but they are too difficult to find in practice. Declaring “ $\sqsubseteq$ ”-minimal stubborn sets as optimal because they are “ $\sqsubseteq$ ”-minimal would not be particularly informative. There is, however, a meaningful non-trivial sense in which “ $\sqsubseteq$ ”-minimal stubborn sets are optimal. It is given in the next definition.

**Definition 10.** *Let  $(E, D, C, “\rightsquigarrow”)$  be a dependency graph. A set  $V \subseteq E \cup D \cup C$  is DG-optimal if and only if  $V \cap E$  is minimal among those sets that are deadlock-preserving for all Petri nets that are respected by  $(E, D, C, “\rightsquigarrow”)$ .*

Before proving that “ $\sqsubseteq$ ”-minimal stubborn sets are DG-optimal, let us comment on why the result is interesting. DG-optimal sets are the best that can be constructed using only information given by the dependency graph. Smaller deadlock-preserving sets may be possible, but to find any, a mechanism stronger than dependency graphs is needed to extract information about the behaviour of the system. Therefore, new ideas are needed to develop such a state space reduction method. Also, if one is puzzled why stubborn set rules take some transition, building the net in the proof reveals the reason.

We now start proving the optimality result. Let  $(E, D, C, “\rightsquigarrow”)$  be a dependency graph and  $V$  a “ $\sqsubseteq$ ”-minimal stubborn set of it. By Theorem 2,  $V \cap E$  is deadlock-preserving for all Petri nets in Definition 10. We will prove its minimality by, for every  $\hat{e} \in V \cap E$ , constructing a Petri net  $(P, T, W, M)$  that  $(E, D, C, “\rightsquigarrow”)$  respects, but  $(V \cap E) \setminus \{\hat{e}\}$  is not deadlock-preserving. Because the proof is long, we proceed via a series of constructions and lemmas.

First we define recursively the sets  $U_i$ ,  $U$ ,  $W_i$ ,  $W$ , and  $X$ , where  $i \in \mathbb{N}$ . Intuitively,  $U$  will be the transitions that may be enabled and the conditions that may be made hold without any enabled transition in  $V$  occurring, with the twist that the transitions are not while the conditions are allowed to be in  $V$ . The set  $W$  will contain the remaining transitions that may be enabled, and  $X$  the transitions of which the dependency graph proves that they are permanently disabled. The construction implies  $U \cap W = U \cap X = W \cap X = \emptyset$ .

- $U_0 = E \setminus V$
- $U_{i+1} = U_i \cup \{d \in D \setminus V \mid d \bullet \subseteq U_i\} \cup \{c \in C \mid c \bullet \cap U_i \neq \emptyset\}$
- $U = U_0 \cup U_1 \cup U_2 \cup \dots$
- $W_0 = \{\hat{e}\}$
- $W_{i+1} = W_i \cup \{d \in D \setminus U \mid d \bullet \subseteq U \cup W_i\} \cup \{v \in (C \cup E) \setminus U \mid v \bullet \cap W_i \neq \emptyset\}$
- $W = W_0 \cup W_1 \cup W_2 \cup \dots$
- $X = (E \cup D \cup C) \setminus (U \cup W)$

The next lemma states an important property of these sets.

**Lemma 1.**  *$X$  satisfies parts 2 and 3 of Definition 8, and  $E \subseteq U \cup W$ .*

*Proof.* We show that  $X$  satisfies part 3 of Definition 8. Let  $d \in X \cap D$ . Then  $d \in D \setminus U$ . If  $d \bullet \subseteq U \cup W$ , then  $d \in W$  by the construction of  $W$ . But this is in contradiction with  $d \in X$ . Therefore,  $d \bullet \not\subseteq U \cup W$ , that is,  $d \bullet \cap X \neq \emptyset$ .

We show next that  $X$  also satisfies part 2. Let  $c \in X \cap C$ . Then  $c \in C \setminus U$  and  $c \notin W$ , so  $c \bullet \cap W = \emptyset$  by the construction of  $W$ . The definition of  $U$  and  $c \notin U$  imply  $c \bullet \cap U = \emptyset$ . So  $c \bullet \subseteq X$ . Let  $e \in X \cap E$ . We conclude  $e \bullet \cap W = \emptyset$  like above with  $c$ . The definition of  $U_0$  and  $e \in E \setminus U$  imply  $e \in V$ . Because  $V$  is a stubborn set we have  $e \bullet \subseteq V$ . Definition 6 implies  $e \bullet \subseteq E \cup D$  and the construction of  $U$  guarantees that  $U \cap V \cap (E \cup D) = \emptyset$ , so  $e \bullet \cap U = \emptyset$ . In conclusion,  $e \bullet \subseteq X$ .

We saw above that  $X \cap E \subseteq V$ , that is,  $X \sqsubseteq V$ . It is immediate from the construction that  $\hat{e} \notin X$ , implying  $X \sqsubset V$ . Thus, and because  $V$  is a “ $\sqsubseteq$ ”-minimal stubborn set,  $X$  cannot be stubborn. So  $X$  must violate part 1 of Definition 8. That is,  $X \cap E = \emptyset$ , giving the claim.  $\square$

To continue the construction, we need two orderings of vertices. The first one, whose “less than” relation is denoted with “ $<$ ”, may be chosen arbitrarily. The second one is defined by  $u < v$  if and only if either  $u$  is in an earlier set in the following list than  $v$ , or they are in the same set and  $u < v$ :

$$\{\hat{e}\}, U_0, U_1 \setminus U_0, U_2 \setminus U_1, \dots, W_1 \setminus W_0, W_2 \setminus W_1, \dots, X.$$

We are now ready to construct the Petri net  $(P, T, W, M)$ . Its transitions are  $T = D \cup E$ . For every  $t \in T$  we introduce a place  $p_t$  such that  $M(p_t) = 1$ ,  $W(p_t, t) = 1$ , and  $p_t$  has no other adjacent arcs. Because of these places, each transition can occur at most once. If  $M_1$  is reached via a transition sequence that contains some transition  $t$  and  $M_2$  is reached via a transition sequence that does not contain  $t$ , then  $M_1 \neq M_2$  because  $M_1(p_t) = 0$  and  $M_2(p_t) = 1$ .

For every  $c \in C$  we introduce a place  $p_c$  such that  $M(p_c) = 0$ . For every  $d \in D$  such that  $d \rightsquigarrow c$  we let  $W(p_c, d) = 1$ . For every  $t \in T$  such that  $c \rightsquigarrow t$  we let  $W(t, p_c) = |\{d \mid d \rightsquigarrow c\}| + 1$ . The place  $p_c$  has no other adjacent arcs. The occurrence of any  $t$  such that  $c \rightsquigarrow t$  marks  $p_c$  with so many tokens that they suffice for one occurrence of each transition that consumes a token from  $p_c$ . The “ $+1$ ” ensures that an arc from  $t$  to  $p_c$  is drawn even if there is no  $d$  such that  $d \rightsquigarrow c$ .

For every  $e \in E$  and  $t \in T$  such that  $t < e$  and  $e \rightsquigarrow t$  we introduce a place  $p_{t,e}$  such that  $M(p_{t,e}) = 1$  and  $W(p_{t,e}, e) = W(p_{t,e}, t) = W(t, p_{t,e}) = 1$ , and  $p_{t,e}$  has no other adjacent arcs. This guarantees that if  $e$  occurs before  $t$ , then  $t$  will never occur, but the occurrence of  $t$  does not affect  $e$  in a similar way. For every  $e \in E$  and  $d \in D$  such that  $e < d$  and  $e \rightsquigarrow d$  we introduce a place  $p_{e,d}$  such that  $M(p_{e,d}) = 1$ ,  $W(p_{e,d}, d) = W(p_{e,d}, e) = W(e, p_{e,d}) = 1$ , and  $p_{e,d}$  has no other adjacent arcs. This completes the construction of the net  $(P, T, W, M)$ .

Our next duty is to show that  $(E, D, C, “\rightsquigarrow”)$  respects  $(P, T, W, M)$ . According to Definition 7, condition vertices that have no input edges are irrelevant. Knowing that, the following lemma gives the claim.

**Lemma 2.** *The dependency graph of  $(P, T, W, M)$  that is constructed as in Example 2 is  $(E, D, C, “\rightsquigarrow”)$  with those  $c \in C$  and their output edges removed that have no input edges.*

*Proof.* The proof mostly consists of easy checks. The only case deserving a comment is  $e_1 \rightsquigarrow e_2$ , where  $e_1 \in E$  and  $e_2 \in E$ . The constructed net fragment

induces also the edge  $e_2 \rightsquigarrow e_1$ . This is, however, what it should do, because of part 4 of Definition 6.  $\square$

The next lemma states that the transitions in  $U \cup W$  can occur in the order specified by “ $\prec$ ”.

**Lemma 3.** *Let  $t_1, t_2, \dots, t_n$  be the elements of  $(U \cup W) \setminus C$  sorted so that  $t_i \prec t_j$  if  $i < j$ . Then  $M [t_1 t_2 \dots t_n]$ .*

*Proof.* Obviously  $\hat{e}$  can occur first. It is easy to check from the construction that when an originally enabled transition occurs, the tokens that it consumes and does not put back are not needed by the transitions that should occur later. When an originally disabled transition occurs, it may consume tokens from input places of other originally disabled transitions. However, when a token was put into such a place, so many tokens were put that every transition that inputs from the place can occur once. Therefore, occurrences of originally disabled transitions do not disable transitions that should occur later.

We still have to show that each originally disabled transition  $d$  is enabled by the time it should occur. If  $d \in U_{i+1} \setminus U_i$ , then all of its empty input places are in  $U_i$ , and each of them has an input transition in  $U_{i-1}$ . So  $d$  is enabled when it is its time to occur. Similar but slightly more complicated reasoning applies when  $d \in W_{i+1} \setminus W_i$ .  $\square$

Let  $M_d$  be the marking that satisfies  $M [t_1 t_2 \dots t_n] M_d$ . Lemma 1 implies that the transitions in  $X$  are permanently disabled. Therefore,  $M_d$  is a deadlock. The next lemma states the last detail that is missing from our proof.

**Lemma 4.** *Let  $M [t] M'$ , where  $t \in V \setminus \{\hat{e}\}$ .  $M_d$  cannot be reached from  $M'$ .*

*Proof.* Obviously  $t$  is enabled in  $M$ , that is,  $t \in E$ . Lemma 1 and the fact that  $U \cap V \cap E = \emptyset$  imply  $t \in W$ . Because  $t \neq \hat{e}$ , we have  $t \notin W_0$ . There is thus some  $i > 0$  such that  $t \in W_i \setminus W_{i-1}$ , and some  $v \in W_{i-1}$  such that  $t \rightsquigarrow v$  and  $v \prec t$ . Because  $t \in E$  we have  $v \in D \cup E$ , that is,  $v$  is a transition. By the construction,  $t$  consumes a token that  $v$  needs, and no transition can put that token back. Therefore, any occurrence sequence that starts at  $M'$  lacks  $v$ , and every marking  $M''$  to which such an occurrence sequence leads has  $M''(p_v) = 1$ . On the other hand,  $M_d(p_v) = 0$ , because  $v \in W_{i-1}$ . So  $M'' \neq M_d$ .  $\square$

We have completed the proof of the following theorem.

**Theorem 4.** “ $\sqsubseteq$ ”-minimal stubborn sets are DG-optimal sets.

## 9 Conclusions

We discussed the effects of different dependency relations and ways of controlling disabled transitions, conjuncts in enabling conditions, and the use of processes or transitions as basic units of constructing stubborn sets. We discussed different sets of states with respect to which dependency relations are defined, and pointed

out that there is no natural “best” dependency relation. We also briefly discussed algorithms. We proved that “ $\sqsubseteq$ ”-minimal stubborn sets are the best that can be obtained without using information other than that in dependency graphs.

We saw that a small change in the method may lead to a dramatic growth in the size of the reduced state space. It seems that state explosion has many sources, and if they are not all kept in control, state spaces explode easily.

Obvious topics for future work are comparison of stubborn sets that are not in the “ $\sqsubseteq$ ”-relation to each other, the usefulness of algorithms that exploit the choices offered by part 3 in Definition 8, the many issues that arise in methods that preserve other properties than deadlocks, and weak stubborn sets.

## References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
2. Esparza, J.: Decidability and Complexity of Petri Net Problems – An Introduction. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 374–428. Springer, Heidelberg (1998)
3. Flanagan, C., Godefroid, P.: Dynamic Partial-Order Reduction for Model Checking Software. In: Proceedings of the 32nd Annual ACM Symposium on Principles of Programming Languages, pp. 110–121 (2005)
4. Geldenhuys, J., Hansen, H., Valmari, A.: Exploring the Scope for Partial Order Reduction. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 39–53. Springer, Heidelberg (2009)
5. Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods. In: Proceedings of CAV 1990. AMS–ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 3, pp. 321–340 (1991)
6. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
7. 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)
8. Peled, D., Valmari, A., Kokkarinen, I.: Relaxed Visibility Enhances Partial Order Reduction. *Formal Methods in System Design* 19, 275–289 (2001)
9. Rauhamaa, M.: A Comparative Study of Methods for Efficient Reachability Analysis. Lic. Tech. Thesis, Helsinki University of Technology, Digital Systems Laboratory, Research Report A-14. Espoo, Finland (1990)
10. Valmari, A.: Error Detection by Reduced Reachability Graph Generation. In: Proceedings of the 9th European Workshop on Application and Theory of Petri Nets, pp. 95–122 (1988)
11. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
12. Varpaaniemi, K.: On the Stubborn Set Method in Reduced State Space Generation. PhD Thesis, Helsinki University of Technology, Digital Systems Laboratory Research Report A-51. Espoo, Finland (1998)

# Efficient Computation of Causal Behavioural Profiles Using Structural Decomposition

Matthias Weidlich<sup>1</sup>, Artem Polyvyanyy<sup>1</sup>, Jan Mendling<sup>2</sup>, and Mathias Weske<sup>1</sup>

<sup>1</sup> Hasso Plattner Institute at the University of Potsdam, Germany  
{Matthias.Weidlich,Artem.Polyvyanyy,Mathias.Weske}@hpi.uni-potsdam.de

<sup>2</sup> Humboldt-Universität zu Berlin, Germany  
Jan.Mendling@wiwi.hu-berlin.de

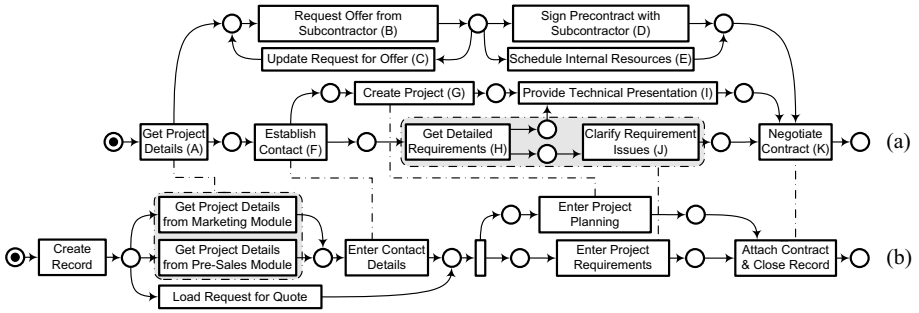
**Abstract.** Identification of behavioural contradictions is an important aspect of software engineering, in particular for checking the consistency between a business process model used as system specification and a corresponding workflow model used as implementation. In this paper, we propose causal behavioural profiles as the basis for a consistency notion, which capture essential behavioural information, such as order, exclusiveness, and causality between pairs of activities. Existing notions of behavioural equivalence, such as bisimulation and trace equivalence, might also be applied as consistency notions. Still, they are exponential in computation. Our novel concept of causal behavioural profiles provides a weaker behavioural consistency notion that can be computed efficiently using structural decomposition techniques for sound free-choice workflow systems if unstructured net fragments are acyclic or can be traced back to S- or T-nets.

## 1 Introduction

Process modelling has recently become one of the most extensively used approaches for capturing business requirements [1]. These requirements are typically refined and modified in an engineering process, resulting in a workflow model and software artefacts. A workflow model often defines activities of the business process model in more detail, neglects steps that are or do not need to be supported by the system, or adjusts behaviour to the specifics of the workflow system. This raises the question to which degree a process model used as specification and a workflow model used as implementation are behaviourally consistent.

Fig. 1 illustrates this problem. Model (a) assumes a business perspective, whereas (b) shows the workflow implementation of the process. Activities (or sets thereof) that correspond to each other are connected by dash-dotted lines. For this paper, we assume that such correspondences are given. They may stem from a system analyst inspecting the models or from automatic matching. Recently, techniques including structural analysis and natural language processing to automatically identify such correspondences have been introduced for the domain of business process models [2,3]. Moreover, techniques known from the





**Fig. 1.** Example of two Petri net process models, (a) focussing on the business perspective, (b) depicting the workflow implementation

area of schema matching [4] can be exploited as activities might be regarded as elements of a process model schema.

In order to reason about the relation between two process models, existing notions of behavioural equivalence might be used as a consistency measure. For instance, bisimulation and trace equivalence assume the set of all traces or the branching structure as essential behavioural characteristics that have to be preserved. However, these notions are computationally hard [5], which is particularly a problem for process models including many activities. Furthermore, these notions only provide information whether behaviour is equivalent or not, but do not describe how strong a deviation is in case of a mismatch.

In this paper, we argue that for the use case of comparing business process models and workflow models, a criterion of behavioural equivalence might be weakened in order to compensate for computational efficiency. We define the notion of a *causal behavioural profile*, which includes dependencies in terms of order, exclusiveness, or causality between pairs of activities. It is computed efficiently using structural decomposition techniques for sound free-choice workflow systems if unstructured net fragments are acyclic or can be traced back to S- or T-nets. We also illustrate how these profiles form the basis of a consistency notion that is weaker than existing notions of behavioural equivalence. Note that proofs not given in this paper are available in a separate technical report [6].

This paper is structured accordingly. Section 2 introduces our formal framework. Causal behavioural profiles are defined in Section 3. Section 4 elaborates on graph decomposition and introduces their application to workflow nets. Their application for computing causal behavioural profiles along with experimental results is presented in Section 5. Finally, Section 6 reviews related work, before Section 7 concludes the paper.

## 2 Preliminaries

We use workflow (WF-) systems [7] as our formal grounding, a class of Petri nets used for process modelling and analysis. Note that Petri net based formalisations

have been presented for (parts of) common process modelling languages, such as BPEL, EPCs, and UML, e.g., [8,9,10]. Based on [7,11], we recall basic definitions.

**Definition 1 (WF-net Syntax)**

- A net is a tuple  $N = (P, T, F)$  with  $P$  and  $T$  as finite disjoint sets of places and transitions, and  $F \subseteq (P \times T) \cup (T \times P)$  as the flow relation. We write  $X = (P \cup T)$  for all nodes. The transitive closure of  $F$  is denoted by  $F^+$ .
- For a node  $x \in X$ ,  $\bullet x := \{y \in X \mid (y, x) \in F\}$ ,  $x \bullet := \{y \in X \mid (x, y) \in F\}$ ,  $\bullet(x \bullet) := \{z \in X \mid y \in X \wedge (x, y) \in F \wedge (z, y) \in F\}$ .
- A tuple  $N' = (P', T', F')$  is a subnet of a net  $N = (P, T, F)$ , if  $P' \subseteq P$ ,  $T' \subseteq T$ , and  $F' = F \cap ((P' \times T') \cup (T' \times P'))$ ;  $N'$  is a partial subnet of  $N$ , if  $F' \subseteq F \cap ((P' \times T') \cup (T' \times P'))$ .
- A net  $N$  is a T-net, if  $\forall p \in P [ |\bullet p| = 1 = |p \bullet| ]$ , and an S-net, if  $\forall t \in T [ |\bullet t| = 1 = |t \bullet| ]$ .
- A net  $N$  is free-choice, iff  $\forall p \in P$  with  $|p \bullet| > 1$  holds  $\bullet(p \bullet) = \{p\}$ .
- A path is a non-empty sequence  $x_1, \dots, x_k$  of nodes,  $k > 1$ , denoted by  $\pi_N(x_1, x_k)$ , which satisfies  $(x_1, x_2), \dots, (x_{k-1}, x_k) \in F$ . We write  $x_i \in \pi_N$ , if  $x_i$  is part of the path  $\pi_N$ . A subpath  $\pi'_N$  of a path  $\pi_N$  is a subsequence that is itself a path. A path  $\pi_N(x_1, x_k)$  is a circuit, if  $(x_k, x_1) \in F$  and no node occurs more than once in the path.
- For a net  $N = (P, T, F)$  and a partial subnet  $N'$  a path  $\pi_N(x_1, x_k)$ ,  $k > 1$  and all  $x_i$  are distinct, of  $N$  is a handle of  $N'$ , iff  $\pi_N \cap (P' \cup T') = \{x_1, x_k\}$ .
- For a net  $N = (P, T, F)$  and two partial subnets  $N'$ ,  $N''$  a path  $\pi_N(x_1, x_k)$  ( $k > 1$  and all  $x_i$  are distinct) of  $N$  is a bridge from  $N'$  to  $N''$ , iff  $\pi_N \cap (P' \cup T') = \{x_1\}$  and  $\pi_N \cap (P'' \cup T'') = \{x_k\}$ .
- A Petri net  $N = (P, T, F)$  is a workflow (WF-) net, iff  $N$  has an initial place  $i \in P$  with  $\bullet i = \emptyset$ ,  $N$  has a final place  $o \in P$  with  $o \bullet = \emptyset$ , and the short-circuit net  $N' = (P, T \cup \{t_c\}, F \cup \{(o, t_c), (t_c, i)\})$  of  $N$  is strongly connected.

Note that we speak of *PP*-, *TT*-, *PT*-, *TP*- handles and bridges, depending on the type (place or transition, respectively) of the initial and the final node of the respective path. Further on, we define semantics for WF-nets according to [7].

**Definition 2 (WF-net Semantics).** Let  $N = (P, T, F)$  be a WF-net with initial place  $i$  and final place  $o$ .

- $M : P \mapsto \mathbb{N}$  is a marking of  $N$ ,  $\mathbb{M}$  denotes all markings of  $N$ .  $M(p)$  returns the number of tokens in place  $p$ .  $[p]$  denotes the marking when place  $p$  contains just one token and all other places contain no tokens.
- For any transition  $t \in T$  and any marking  $M \in \mathbb{M}$ ,  $t$  is enabled in  $M$ , denoted by  $(N, M)[t]$ , iff  $\forall p \in \bullet t [ M(p) \geq 1 ]$ .
- Marking  $M'$  is reached from  $M$  by firing of  $t$ , denoted by  $(N, M)[t](N, M')$ , such that  $M' = M - \bullet t + t \bullet$ , i.e., one token is taken from each input place of  $t$  and one token is added to each output place of  $t$ .
- A firing sequence of length  $n \in \mathbb{N}$  is a function  $\sigma : \{0, \dots, n-1\} \mapsto T$ . For  $\sigma = \{(0, t_x), \dots, (n-1, t_y)\}$ , we also write  $\sigma = t_0, \dots, t_{n-1}$ .
- For any two markings  $M, M' \in \mathbb{M}$ ,  $M'$  is reachable from  $M$  in  $N$ , denoted by  $M' \in [N, M_i]$ , if there exists a firing sequence  $\sigma$  leading from  $M$  to  $M'$ .

- A net system, or a system, is a pair  $(N, M_i)$ , where  $N$  is a net and  $M_i$  is the initial marking of  $N$ . A WF-system is a pair  $(N, M_i)$ , where  $N$  is a WF-net with initial place  $i$  and  $M_i = [i]$ .

Note that the *final marking* is denoted by  $M_o$ . Without stating it explicitly, we assume a net of a system to be defined as  $N = (P, T, F)$ . Moreover, when the context is clear, we refer to WF-systems and short-circuit nets as WF-nets. Finally, we recall the *soundness* property, which requires WF-systems (1) to always terminate, and (2) to have no dead transitions (proper termination is implied for WF-systems) [12].

**Definition 3 (Liveness, Boundedness, Soundness)**

- A system  $(N, M_i)$  is *live*, iff for every reachable marking  $M \in [N, M_i)$  and  $t \in T$ , there exists a marking  $M' \in [N, M)$  such that  $(N, M')[t]$ .
- A system  $(N, M_i)$  is *bounded*, iff the set  $[N, M_i)$  is finite.
- A WF-system  $(N, M_i)$  is *sound*, iff the short-circuit system  $(N', M_i)$  is live and bounded.

### 3 The Notion of a Causal Behavioural Profile

This section introduces causal behavioural profiles. They are based on the notion of behavioural profiles, which we recall in Section 3.1. We introduced these profiles in an earlier work [13] to reason on execution ordering constraints only. Thus, optionality of transition execution or causality between transitions is not captured. These aspects are addressed by the novel concept of a causal behavioural profile introduced in Section 3.2. Section 3.3 discusses our concepts in the light of existing behavioural models defined for Petri nets. Finally, we discuss the application of causal behavioural profiles for consistency checking in Section 3.4.

#### 3.1 Execution Order Constraints: The Behavioural Profile

*Behavioural profiles* aim at capturing behavioural aspects in terms of order constraints of a process in a fine-grained manner [13]. They are grounded on the set of possible firing sequences of a WF-system and the notion of *weak order*.

**Definition 4 (Weak Order).** Let  $(N, M_i)$  be a WF-system. A pair  $(x, y)$  is in the weak order relation  $\succ \subseteq T \times T$ , iff there exists a firing sequence  $\sigma = t_1, \dots, t_n$  with  $(N, M_i)[\sigma]$ ,  $j \in \{1, \dots, n-1\}$ ,  $j < k \leq n$ , for which holds  $t_j = x$  and  $t_k = y$ .

Thus, two transitions  $t_1, t_2$  are in weak order, if there exists a firing sequence reachable from the initial marking in which  $t_1$  occurs before  $t_2$ . Depending on how two activities of a process model are related by weak order, we define three relations forming the behavioural profile.

**Definition 5 (Behavioural Profile).** Let  $(N, M_i)$  be a WF-system. A pair  $(x, y) \in (T \times T)$  is in at most one of the following relations:

- The strict order relation  $\rightsquigarrow$ , if  $x \succ y$  and  $y \neq x$ .
- The exclusiveness relation  $+$ , if  $x \neq y$  and  $y \neq x$ .
- The interleaving order relation  $\parallel$ , if  $x \succ y$  and  $y \succ x$ .

Given a set  $T' \subseteq T$ , the set of all relations  $BP_{T'} = \{\rightsquigarrow, +, \parallel\}$  defined over  $T' \times T'$  is the *behavioural profile* of  $(N, M_i)$  for  $T'$ .

Computing the behavioural profile for all transitions of the system (a) in Fig. 1, for instance, it holds  $C \rightsquigarrow E$  as there exists no firing sequence, such that  $E$  occurs before  $C$ . However, strict order does not imply the actual occurrence. That is, there are firing sequences containing only one of the two transitions, or even none of them. It holds  $D + E$  as both transitions will never occur in a single firing sequence and  $B \parallel G$  as both transitions can occur in any order. Note that the three relations are mutually exclusive and (together with *reversed* strict order) partition the Cartesian product of transitions over which they are defined [13]. With respect to itself, a transition is either in the exclusive relation (if it can occur at most once, e.g.,  $D + D$ ) or in the interleaving order relation (if it can occur more than once, e.g.,  $B \parallel B$ ).

### 3.2 Occurrence Constraints: The Causal Behavioural Profile

Behavioural profiles as introduced above relate pairs of transitions according to their *order* of potential occurrence. It is important to see that for the case of validating a workflow implementation against a process model specification, the relation associating corresponding transitions of both models to each other is typically partial. That is, certain transitions of one model are without counterpart in the other model, cf., Fig. 1

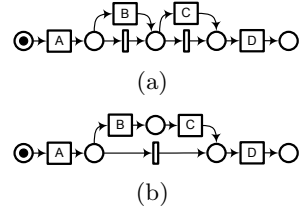


Fig. 2. Optionality

As a consequence, information on ordering constraints is not sufficient to draw conclusions on optionality and causality of transition occurrences.

*Optionality* of a transition is given, if there is a firing sequence leading from the initial to the final marking of the system that does not contain the transition. Optionality can be lifted from single transitions to sets of transitions. A set of transitions is considered to be jointly optional, if any firing sequence from the initial to the final marking contains all or none of the transitions. As illustrated by Fig. 2(a) and Fig. 2(b) this property cannot be derived from the knowledge about optionality of single transitions. In both systems,  $B$  and  $C$  are optional, but only in Fig. 2(b) the set  $\{B, C\}$  is jointly optional.

Closely related to optionality is *causality*, which requires that one transition can only occur after the occurrence of another transition. Thus, causality comprises two aspects, a certain *order* of occurrences and a *causal coupling* of occurrences. While the former is addressed by the behavioural profile in terms

of the strict order relation, the latter is not captured. For instance,  $B$  is a cause of  $C$  in Fig. 2(b), but not in Fig. 2(a). Note that two transitions in interleaving order cannot show causality according to our definition. For both systems in Fig. 3, it holds  $B||C$ , as there is no distinct order relation between *all* occurrences of both transitions. Thus, interleaving order is interpreted as the absence of any dependency regarding the order of occurrence. Thus, it is reasonable to define causality as a dependency between *all* occurrences of two transitions, instead of considering causal dependencies between their *single* occurrences (cf., *response/leads-to* dependencies [14]). There is no causality between  $B$  and  $C$  in either system in Fig. 3.

In order to cope with the aforementioned aspects, we introduce the co-occurrence relation and the causal behavioural profile. Two transitions are co-occurring, if any firing sequence from the initial to the final marking that contains the first transition contains also the second transition.

**Definition 6 (Causal Behavioural Profile).** *Let  $(N, M_i)$  be a WF-system.*

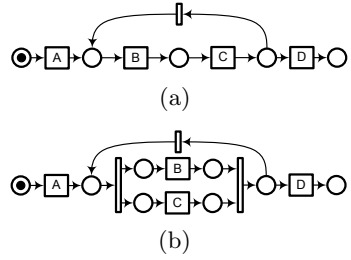
- A pair  $(x, y) \in (T \times T)$  is in the co-occurrence relation  $\gg$ , if for all firing sequences  $\sigma$  with  $(N, M_i)[\sigma](N, M_o)$ , it holds  $x \in \sigma \Rightarrow y \in \sigma$ .
- Given a set  $T' \subseteq T$ , the set of all relations  $CBP_{T'} = \{\rightsquigarrow, +, ||, \gg\}$  defined over  $T' \times T'$  is the causal behavioural profile of  $(N, M_i)$  for  $T'$ .

Trivially, it holds  $t \gg t$  for all  $t \in T$ . We derive optionality and causality as follows. A single transition  $t \in T$  is optional, if  $t_i \rightsquigarrow t$  for some  $t_i \in i \bullet$  with  $i$  as the initial place. A set  $T_1 \subseteq T$  of transitions is optional, if all transitions themselves are optional and they are pairwise co-occurring to each other ( $(T_1 \times T_1) \subseteq \gg$ ).

Further on, there is a causal dependency between two transitions  $t_1, t_2 \in T$ , if they are in strict order ( $t_1 \rightsquigarrow t_2$ ) and occurrence of the first implies occurrence of the second ( $t_1 \gg t_2$ ). Note that, in contrast to the behavioural profile, the *causal* behavioural profile differs for both systems in Fig. 2.

### 3.3 Relation to Existing Behavioural Models

There is a large body of research on behavioural relations for formal models specifying dynamic systems in general, and for Petri nets in particular. Focussing on the order of occurrence, the relations proposed in [15] for workflow mining are close to our relations, yet different. We base our definitions on the notion of an *indirect* weak order dependency, whereas the relations in [15] are grounded on a *direct* sequential order. As a result, the notion of exclusiveness is restricted to ‘*pairs of transitions that never follow each other directly*’ [15], whereas we capture exclusiveness for transitions that might occur at different stages of a firing sequence. While the notion of direct sequential order is appropriate for



**Fig. 3.** No causality for transitions  $(B, C)$  in a cycle

workflow mining, it leads to undesired effects in our setting. Consider, for instance, transitions  $G$  and  $K$  of model (a) in Fig. 1. They are exclusive according to the relations proposed in [15], whereas their counterparts in model (b) are in a sequential order. The behavioural profile, in turn, yields equal relations in both models. The respective transitions are in strict order in both models, (a) and (b).

Obviously, the well-known notions of *conflicting* and *concurrent* transitions are related to our *observed* relations as well. In a sound free-choice WF-system, two transitions in conflict, which are not part of a common control flow cycle will be exclusive in the behavioural profile. This follows from Lemma 3 in [13] and the fact that sound free-choice WF-systems are safe (a place carries at most one token in all markings, cf., Lemma 1 in [24]). Similarly, all transitions that are enabled concurrently in some reachable marking (cf., the concurrency relation [16]) are in interleaving order in the behavioural profile.

In order to cope with concurrency and the interleaving problem, the *unfolding* of a Petri net (or its prefix, respectively) might be exploited for behaviour analysis [17,18]. That is, a *true concurrent* model is created in which a transition (i.e., an *event*) corresponds to a certain *occurrence* of a transition in the original net. Events can be related as being in a *weak causal predecessor*, *conflict*, or *concurrency* relation. While these relations resemble the relations of our casual behavioural profile, they are defined for transition occurrences instead of transitions. Thus, we might derive our relations by lifting these relations to the level of transitions again. For instance, if all events representing two transitions are in conflict in the (potentially infinite) unfolding, both transitions are exclusive according to the behavioural profile. However, an algorithm for the derivation of causal behavioural profiles from the prefix of an unfolding is beyond the scope of this paper. Usage of unfoldings is also inappropriate w.r.t. the class of systems we address in this paper, as the construction of unfoldings is computationally much harder than the approach introduced in the remainder of this paper.

With respect to common notions of behavioural equivalence, we see that two WF-systems with equal causal profiles are not necessarily trace equivalent. For instance, both systems in Fig. 3 have the same causal profile, whereas they are not trace equivalent. Evidently, the same holds true for bisimulation equivalences, as the profile neglects the branching structure of a system. However, it is easy to see that trace equivalence of two WF-systems implies equivalence of their causal behavioural profiles for all transitions, as all behavioural relations formulate statements about the existence of firing sequences.

### 3.4 Application of Causal Behavioural Profiles

We motivated the definition of causal behavioural profiles with the need for a notion of behavioural consistency that enables analysis of related process models in an efficient manner. Under the assumption of an alignment relation between transitions of two WF-systems, we define a degree of consistency as follows.

**Definition 7 (Degree of Consistency).** *Let  $(N_1, M_{i_1})$  and  $(N_2, M_{i_2})$  be two WF-systems and  $\sim \subseteq T_1 \times T_2$  a correspondence relation with  $\sim \neq \emptyset$ .*

- The set  $T_1^\sim = \{t_1 \in T_1 \mid \exists t_2 \in T_2 [t_1 \sim t_2]\}$  contains all aligned transitions of  $(N_1, M_{i_1})$ .  $T_2^\sim$  is defined analogously.
  - With  $\mathcal{R}_1$  and  $\mathcal{R}_2$  as the relations of the causal behavioural profile for the WF-systems, the set  $CT_1^\sim \subseteq (T_1^\sim \times T_1^\sim)$  contains all consistent transition pairs  $(t_x, t_y)$ , such that
    - if  $t_x = t_y$ , then  $\forall t_s \in T_2^\sim$  with  $t_x \sim t_s$  it holds  $t_x \mathcal{R}_1 t_x \Rightarrow t_s \mathcal{R}_2 t_s$ ,
    - if  $t_x \neq t_y$ , then  $\forall t_s, t_t \in T_2^\sim$  with  $t_s \neq t_t$ ,  $t_x \sim t_s$ , and  $t_y \sim t_t$  it holds either  $t_x \mathcal{R}_1 t_y \Rightarrow t_s \mathcal{R}_2 t_t$  or  $t_x \sim t_t$  and  $t_y \sim t_s$ .
- The set  $CT_2^\sim$  is defined analogously.
- The degree of consistency of  $\sim$  is defined as  $\mathcal{D}^\sim = \frac{|CT_1^\sim| + |CT_2^\sim|}{|(T_1^\sim \times T_1^\sim)| + |(T_2^\sim \times T_2^\sim)|}$ .

The general idea behind this degree can be summarised as follows. For each pair of transitions, for which there are corresponding transitions in the other model, we check whether they share the same constraints. Since there can be complex 1:n correspondences as in Fig. 11, we have to count these correspondences from the perspective of each model. Applying this degree to the scenario in Fig. 11, we see that the order of potential occurrence is preserved for all aligned transitions. However, transition (A) is mandatory in model (a), whereas its counterparts are optional in model (b). Consequently, causality between transition (A) and, for instance, transition (K) is not preserved in model (b) either, which is taken into account in the causal behavioural profile. For our example, the degree of consistency is  $\mathcal{D}^\sim = \frac{28+27}{36+36} \approx 0.76$ , as both models (a) and (b) contain six transitions with correspondences yielding 36 transition pairs in the profile, while the profile relations are preserved for 28 (or 27, respectively) pairs.

The degree of consistency as defined above shows the characteristics of a semimetric for the comparison of two causal behavioural profiles. That is, the degree of consistency is non-negative and symmetric measure that equals one (or zero if it is subtracted from one, respectively), if and only if both profiles are equal. For the assessment of two profiles, however, the degree of consistency is not a metric as it does not satisfy the triangle inequality. That is due to the fact that the degree is a criterion for the quality of an alignment, i.e., a set of a correspondences. Hence, it is normalised by the size (the number of transitions) of the alignment but independent of the size of the respective WF-systems and, therefore, causal behavioural profiles. Still, we see that the relations of the causal behavioural profile are transitive in the sense that equal relations between a first and a second model, and the second and a third model imply the equivalence for the relations between the first and the third model. Thus, triangle inequality holds for the comparison of the degree of consistency of different alignments when considering solely those pairs of transitions that are part of all alignments.

For our proposal of assessing the consistency between business process models and their implementation as a workflow model, we got positive feedback from process analysts. Currently, we are evaluating the results of an empirical study that relates our degree of consistency to the consistency perception of process experts in a broader setting. Here, preliminary findings confirm a good approximation of perceived consistency by our degree. Clearly, there is a need for a multitude of consistency criteria in order to be able to graduate consistency

requirements for a concrete setting. Nevertheless, an interval scale and efficient computation methods have to be seen as core requirements on such notions.

It is worth to mention that we already showed how behavioural profiles can be applied to support change propagation between related process models [19].

## 4 Graph Decomposition Techniques for WF-Systems

First, Section 4.1 introduces the Refined Process Structure Tree (RPST), a structural decomposition technique for workflow graphs. Second, Section 4.2 enriches the RPST for WF-systems with behavioural annotations.

### 4.1 The Refined Process Structure Tree

The RPST [20,21] is a technique for detecting the structure of a workflow graph. A workflow graph can be parsed into a hierarchy of *fragments* with a single entry and a single exit, such that the RPST is a containment hierarchy of *canonical fragments* of the graph. The RPST is unique for a given workflow graph and can be computed in linear time [20,21]. Although the RPST has been introduced for workflow graphs, the technique can be applied to other graph based behavioural models such as WF-systems in a straight-forward manner. Basic terms of the RPST are defined for WF-nets as follows.

#### Definition 8 (Edges, Entry, Exit, Canonical Fragment)

Let  $N = (P, T, F)$  be a WF-net.

- For a node  $x \in X$  of a net  $N = (P, T, F)$ ,  $in_N(x) = \{(n, x) \mid n \in \bullet x\}$  are its incoming edges and  $out_N(x) = \{(x, n) \mid n \in x \bullet\}$  are its outgoing edges.
- A node  $x \in X'$  of a connected subnet  $N' = (P', T', F')$  of a net  $N$  is a boundary node, if  $\exists e \in in_N(x) \cup out_N(x) [ e \notin F' ]$ . If  $x$  is a boundary node, it is an entry of  $N'$ , if  $in_N(x) \cap F' = \emptyset$  or  $out_N(x) \subseteq F'$ , or an exit of  $N'$ , if  $out_N(x) \cap F' = \emptyset$  or  $in_N(x) \subseteq F'$ .
- Any connected subnet  $\omega$  of  $N$ , is a fragment, if it has exactly two boundary nodes, one entry and one exit denoted by  $\omega_{\triangleleft}$  and  $\omega_{\triangleright}$ , respectively.

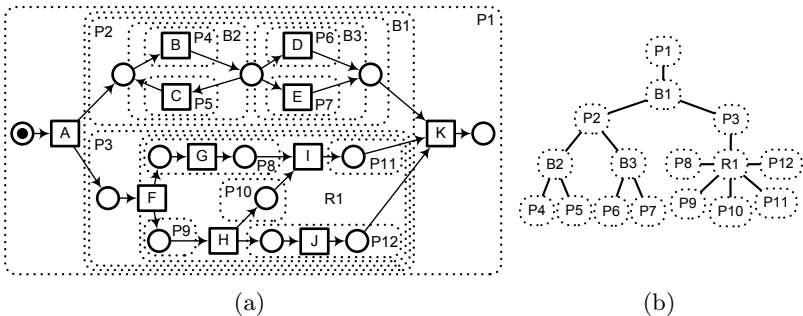


Fig. 4. (a) A WF-system and its canonical fragments, (b) the RPST of (a)



- A *fragment* is place bordered if its boundary nodes are places.
- A *fragment* is transition bordered if its boundary nodes are transitions.
- A *fragment*  $\omega = (P_\omega, T_\omega, F_\omega)$  is canonical in a set of all fragments  $\Sigma$  of  $N$ , iff  $\forall \gamma = (P_\gamma, T_\gamma, F_\gamma) \in \Sigma [ \omega \neq \gamma \Rightarrow (F_\omega \cap F_\gamma = \emptyset) \vee (F_\omega \subset F_\gamma) \vee (F_\gamma \subset F_\omega) ]$ .

Fig. 4 exemplifies the RPST for the WF-system from Fig. 1(a). Fig. 4(a) illustrates its canonical fragments, each of them formed by a set of edges enclosed in or intersecting the region with a dotted border. Fig. 4(b) provides an alternative view, where each node represents a canonical fragment and edges hint at containment relation of fragments. Observe that one obtains a tree structure—the RPST. For instance, fragment  $B1$  has two boundary transitions: entry  $A$  and exit  $K$ , is contained in fragment  $P1$ , and contains fragments  $P2$  and  $P3$ .

If the RPST is computed for a *normalized* workflow graph, i.e., a workflow graph that does not contain nodes with multiple incoming and multiple outgoing edges, each canonical fragment can be classified to one out of four structural classes [21,22]: A *trivial* ( $T$ ) fragment consists of a single edge. A *polygon* ( $P$ ) represents a sequence of nodes (fragments). A *bond* ( $B$ ) stands for a collection of fragments that share common boundary nodes. Any other fragment is a *rigid* ( $R$ ). Note that we use fragment names that hint at their structural class, e.g.,  $R1$  is a rigid fragment. Every workflow graph can be normalized by performing a node-splitting pre-processing step, illustrated for WF-nets in Fig. 5. The WF-system in Fig. 4(a) is normalized.

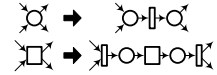


Fig. 5. Node-splitting

## 4.2 An Annotated RPST: The WF-Tree

The structural patterns derived by the RPST can be related to behavioural properties of the underlying WF-system. In this section, we concretise RPST fragments by annotating them with behavioural characteristics. We refer to the containment hierarchy of annotated canonical fragments of a WF-system as the RPST with behavioural annotations, or *WF-tree* for short. The WF-tree is defined for sound free-choice WF-systems. It is well-known that the free-choice and soundness properties are required to derive behavioural statements from the structure of a system, as both together imply a tight coupling of syntax and semantics (cf., [23,24]).

**Definition 9 (WF-Tree).** Let  $(N, M_i)$  be a sound free-choice WF-system. The RPST with behavioural annotations, the WF-Tree of  $N$ , is a tuple  $\mathcal{T}_N = (\Omega, \chi, t, b)$ , where:

- $\Omega$  is a set of all canonical fragments of  $N$ ,
- $\chi : \Omega \rightarrow \mathcal{P}(\Omega)$  is a function that assigns to fragment its child fragments,
- $t : \Omega \rightarrow \{T, P, B, R\}$  is a function that assigns a type to a fragment,
- $b : \Omega_B \rightarrow \{B_\circ, B_\circ, L\}$ ,  $\Omega_B = \{\omega \in \Omega \mid t(\omega) = B\}$ , is a function that assigns a refined type to a bond fragment, where  $B_\circ$ ,  $B_\circ$ , and  $L$  types stand for place bordered, transition bordered, and loop bonds, respectively.

Further on, we define auxiliary concepts for the WF-tree.

**Definition 10 (Parent, Child, Root, Ancestor, Descendant, LCA, Path).**

Let  $\mathcal{T}_N = (\Omega, \chi, t, b)$  be the WF-tree.

- For any fragment  $\omega \in \Omega$ ,  $\omega$  is a parent of  $\gamma$  and  $\gamma$  is a child of  $\omega$ , if  $\gamma \in \chi(\omega)$ . By  $\chi^+$  we denote the transitive closure of  $\chi$ .
- The fragment  $\omega \in \Omega$  is a root of  $\mathcal{T}$ , denoted by  $\omega_r$ , if it has no parent.
- The partial function  $\rho : \Omega \setminus \{\omega_r\} \rightarrow \Omega$  assigns parents to fragments.
- For any fragment  $\omega \in \Omega$ ,  $\omega$  is an ancestor of  $\vartheta$  and  $\vartheta$  is a descendant of  $\omega$ , if  $\vartheta \in \chi^+(\omega)$ .
- For any two fragments  $\{\omega, \gamma\} \in \Omega$  their lowest common ancestor (LCA), denoted by  $\text{lca}(\omega, \gamma)$ , is the shared ancestor of  $\omega$  and  $\gamma$  that is located farthest from the root of the WF-tree. By definition,  $\text{lca}(\omega, \omega) = \omega$ .
- For any fragment  $\omega_0 \in \Omega$  and its descendant  $\omega_n \in \Omega$ , a downward path from  $\omega_0$  to  $\omega_n$ , denoted by  $\pi_{\mathcal{T}}(\omega_0, \omega_n)$ , is a sequence  $(\omega_0, \omega_1, \dots, \omega_n)$ , such that  $\omega_i$  is a parent of  $\omega_{i+1}$  for all  $i \in \mathbb{N}_0$ . In addition,  $\pi_{\mathcal{T}}(\omega_0, \omega_n, i) = \omega_i$  and  $\pi_{\mathcal{T}}\{\omega_0, \omega_n\}$  is a set which contains all fragments of  $\pi_{\mathcal{T}}(\omega_0, \omega_n)$ .

Fig. 6 shows the WF-tree of the WF-system from Fig. 4(a). Note that trivial fragments are not visualised. The WF-tree is isomorphic to the RPST of the WF-system, cf., Fig. 4(b). Given the RPST, adding the behavioural annotation is a trivial task for most fragments, except of the following cases: A bond fragment  $\gamma = (P_\gamma, T_\gamma, F_\gamma) \in \text{dom}(b)$  of  $\mathcal{T}_N = (\Omega, \chi, t, b)$  is assigned the  $L$  type, if  $\gamma_{\triangleleft} = \omega_{\triangleright}$  with  $\omega$  being a child of  $\gamma$ . Otherwise,  $b(\gamma) = B_o$  if  $\gamma_{\triangleleft} \in P_\gamma$ , or  $b(\gamma) = B_o$  if  $\gamma_{\triangleleft} \in T_\gamma$ .

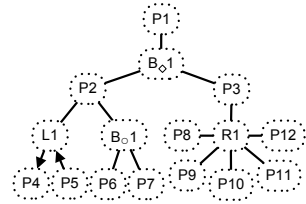


Fig. 6. The WF-tree

Children of a polygon fragment are arranged with respect to their execution order. A partial function  $\text{order} : \Omega' \rightarrow \mathbb{N}_0$ ,  $\Omega' = \{\omega \in \Omega \setminus \{\omega_r\} \mid t(\rho(\omega)) = P\}$  assigns to children of polygon fragments their respective order positions;  $\text{order}(\omega) = 0$ , if  $\omega_{\triangleleft} = \gamma_{\triangleleft}$  with  $\gamma = \rho(\omega)$  being the parent, and  $\text{order}(\omega) = i$ ,  $i \in \mathbb{N}$ , if  $\omega_{\triangleleft} = \vartheta_{\triangleright}$  for some  $\vartheta \in \Omega$ , such that  $\text{order}(\vartheta) = i - 1$ . Observe that the orders of two nodes are only comparable if they share a common parent. For instance, in Fig. 6,  $\text{order}(L1) = 1$  and  $\text{order}(B_o1) = 2$ . This means that fragment  $L1$  is always executed before fragment  $B_o1$  inside of polygon  $P2$ . The layout of child fragments of polygon fragments hints at their order relations.

Children of a loop fragment are classified as *forward* ( $\Rightarrow$ ) or *backward* ( $\Leftarrow$ ). A partial function  $\ell : \Omega'' \rightarrow \{\Leftarrow, \Rightarrow\}$  with  $\Omega'' = \{\omega \in \Omega \setminus \{\omega_r\} \mid b(\rho(\omega)) = L\}$  assigns an orientation to children of loop fragments.  $\ell(\omega) = \Rightarrow$  if  $\omega_{\triangleleft} = \gamma_{\triangleleft}$  with  $\gamma = \rho(\omega)$ , otherwise  $\ell(\omega) = \Leftarrow$ . In Fig. 6,  $P4$  and  $P5$  are forward and backward fragments, respectively, which is visualised by the direction of edges.

We introduce two lemmas that prove the completeness of the codomain of function  $b$  by showing that a bond fragment is either place or transition bordered, and that each loop fragment is place bordered. Note that a rigid fragment bordered with a place and a transition can still be free-choice and sound (see [25]).

**Lemma 1.** *Let  $\mathcal{T}_N = (\Omega, \chi, t, b)$  be the WF-tree of a sound free-choice WF-system  $(N, M_i)$ ,  $N = (P, T, F)$ . No bond fragment  $\omega \in \Omega$ ,  $t(\omega) = B$ , has  $\{p, t\}$  boundary nodes, where  $p \in P$  and  $t \in T$ .*

*Proof.* Assume  $\omega$  is a bond fragment with  $\{p, t\}$  boundary nodes. There exists a circuit  $\Gamma$  in a short-circuit net of  $N$  that contains  $\{p, t\}$ . Let  $\Gamma_\omega$  be a subpath of  $\Gamma$  inside  $\omega$ . There exists a child fragment  $\gamma$  of  $\omega$  that contains  $\Gamma_\omega$ . A bond fragment has  $k \geq 2$  child fragments, cf., [21][22]. Let  $\vartheta$  be a child of  $\omega$ ,  $\vartheta \neq \gamma$ . We distinguish two cases:

- Let  $H$  be a path from  $p$  to  $t$  contained in  $\vartheta$ .  $H$  is a PT-handle of  $\Gamma$ . In a live and bounded free-choice system,  $H$  is bridged to  $\Gamma_\omega$  through a TP-bridge  $K$ , cf., Proposition 4.2 in [26]. This implies that  $\vartheta = \gamma$ ; otherwise bond fragment  $\omega$  contains path  $K$  that is not inside of a single child fragment, cf., [22][21]. Thus,  $\omega$  has a single child fragment, a contradiction with the assumption of  $\omega$  being a bond fragment.
- Let  $H$  be a path from  $t$  to  $p$  contained in  $\vartheta$ .  $H$  is a TP-handle of  $\Gamma$ . In a live and bounded free-choice system, no circuit has TP-handles, cf., Proposition 4.1 in [26], which yields a contradiction with our assumptions.  $\square$

**Lemma 2.** *Let  $\mathcal{T}_N = (\Omega, \chi, t, b)$  be the WF-tree of a sound free-choice WF-system,  $(N, M_i)$ ,  $N = (P, T, F)$ . A loop fragment  $\omega = (P_\omega, T_\omega, F_\omega) \in \Omega$ ,  $b(\omega) = L$ , is place bordered, i.e.,  $\{\omega_\triangleleft, \omega_\triangleright\} \in P$ .*

*Proof.* Because of Lemma 1,  $\omega$  is either place or transition bordered. Assume  $\omega$  is transition bordered. There exists place  $p$  such that  $p \in \bullet\omega_\triangleleft \cap P_\omega$ ,  $M_i(p) = 0$ . Transition  $\omega_\triangleleft$  is enabled if there exists a marking  $M \in [(N, M_i)]$  with  $M(p) > 0$ . Since  $\omega$  is a connected subnet, for all  $t \in T_\omega \setminus \{\omega_\triangleleft, \omega_\triangleright\}$  all edges are in  $\omega$ , i.e.,  $(in_N(t) \cup out_N(t)) \subseteq F_\omega$ . Thus, every path from  $i$  to  $p$  visits  $\omega_\triangleleft$ . Thus,  $M(p) > 0$  is only possible, if  $\omega_\triangleleft$  has fired before. We reached a contradiction. Transition  $\omega_\triangleleft$  is never enabled and  $N$  is not live, and hence, not sound. Since any loop fragment is not transition bordered, it is place bordered (Lemma 1).  $\square$

For sound free-choice WF-systems, the WF-tree can be derived efficiently.

**Corollary 1.** *The following problem can be solved in linear time. Given a sound free-choice WF-system, to compute its WF-tree.*

*Proof.* Given a workflow graph, its RPST can be computed in time linear to the number of edges of the graph [20][21]. The number of canonical fragments in the RPST is linear to the number of edges in the workflow graph [21][27][28]. Given the RPST of a WF-system, we iterate over all bond fragments and assign the behavioural annotations. Here, it suffices to check the type of the entry node, either a place or transition, and to determine whether the entry is also the exit of a child fragment. That can be decided in constant time for each fragment. Finally, child fragments of a polygon can be ordered in linear time. We introduce a hash function that returns a child fragment with the given node as an entry and iterate over the children of the polygon.  $\square$

## 5 Efficient Computation of Causal Behavioural Profiles

This section shows how a WF-tree is applied to compute the causal behavioural profile. Section 5.1 introduces the approach for transition pairs that do not require analysis of rigid fragments. Afterwards, we discuss analysis of rigid fragments in Section 5.2 and present experimental performance results in Section 5.3.

### 5.1 Computation without Analysis of Rigid Fragments

For the computation of the causal behavioural profile for a pair of transitions, we assume that each transition has one incoming and one outgoing flow arc. If this is not the case, we apply the pre-processing illustrated in Fig. 5, which preserves the behaviour of the system (cf., [29]) and, therefore, does not change the causal behavioural profile. Given a pre-processed WF-system  $(N, M_i)$  with  $N = (P, T, F)$  and its WF-tree  $\mathcal{T}_N = (\Omega, \chi, t, b)$ , each transition  $t \in T$  is a boundary node of at most two trivial fragments of  $\mathcal{T}_N$ . Thus, it suffices to show how the behavioural relations are determined for the entries of two trivial fragments.

Our computation is based on two elementary properties of free-choice sound WF-systems. If  $(N, M_i)$  is free-choice and sound, it is safe (cf., Lemma 1 in [24]), i.e.,  $\forall p \in P, M(p) < 2$  in all reachable markings. Thus, a single transition cannot be enabled concurrently with itself.

In addition, if  $(N, M_i)$  is free-choice and sound, the existence of a path  $\pi_N(x, y)$  between places  $x$  and  $y$  implies the existence of a firing sequence containing all transitions on  $\pi_N(x, y)$  (cf., Lemma 4.2 in [23]). While the implication actually requires the marking  $M_y = [y]$  to be a home marking (a marking reachable from every marking that is reachable from the initial state), it can be lifted to all home markings with  $M_y(y) > 0$ . Due to soundness of the system  $(N, M_i)$ , the short-circuit system  $(N', M_i)$  is live and bounded, such that all markings  $M \in [N, M_i]$  are home markings in  $(N', M_i)$ . Thus, all markings  $M_y(y) > 0$  are reachable from markings  $M_x(x) > 0$ , if  $M_y, M_x \in [N', M_i]$ .

In the absence of rigid fragments on certain paths, the execution ordering relations and the co-occurrence relation of the causal behavioural profile are computed as follows. Proofs of both propositions can be found in [6].

**Proposition 1.** *Let  $\mathcal{T}_N = (\Omega, \chi, t, b)$  be the WF-tree and  $\alpha, \beta \in \Omega$  two trivial fragments. Let  $\gamma = lca(\alpha, \beta)$  and  $\forall \omega \in \pi_{\mathcal{T}}\{\omega_r, \gamma\} [t(\omega) \neq R]$ .*

1. *If  $\alpha = \beta$ , then  $\alpha_{\triangleleft} \parallel \beta_{\triangleleft}$ , iff  $\exists \omega \in \pi_{\mathcal{T}}\{\omega_r, \gamma\} [b(\omega) = L]$ . Otherwise,  $\alpha_{\triangleleft} + \beta_{\triangleleft}$ .*
2. *If  $\alpha \neq \beta$ ,*
  - *$\alpha_{\triangleleft} \rightsquigarrow \beta_{\triangleleft}$ , iff (1)  $t(\gamma) = P \wedge order(\pi_{\mathcal{T}}(\gamma, \alpha, 1)) < order(\pi_{\mathcal{T}}(\gamma, \beta, 1))$ , and (2)  $\forall \omega \in \pi_{\mathcal{T}}\{\omega_r, \gamma\} [b(\omega) \neq L]$ .*
  - *$\alpha_{\triangleleft} + \beta_{\triangleleft}$ , iff (1)  $b(\gamma) = B_{\circ}$ , and (2)  $\forall \omega \in \pi_{\mathcal{T}}\{\omega_r, \gamma\} [b(\omega) \neq L]$ .*
  - *$\alpha_{\triangleleft} \parallel \beta_{\triangleleft}$ , iff (1)  $b(\gamma) \in \{B_{\circ}, L\}$ , or (2)  $\exists \omega \in \pi_{\mathcal{T}}\{\omega_r, \gamma\} [b(\omega) = L]$ .*

**Proposition 2.** *Let  $\mathcal{T}_N = (\Omega, \chi, t, b)$  be the WF-tree and  $\alpha, \beta \in \Omega$  two trivial fragments,  $\alpha \neq \beta$ . Let  $\gamma = lca(\alpha, \beta)$ ,  $\Pi = \pi_{\mathcal{T}}\{\gamma, \beta\}$ , and  $\forall \omega \in \Pi [t(\omega) \neq R]$ . Then,  $\alpha_{\triangleleft} \gg \beta_{\triangleleft}$ , iff for all  $\omega \in (\Pi \setminus \{\beta\})$  one of the following conditions holds:*

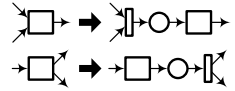


Fig. 7. Pre-processing

1.  $t(\omega) = P$ ,
2.  $t(\omega) = B$  and  $b(\omega) = B_{\circ}$ , or
3.  $t(\omega) = B$ ,  $b(\omega) = L$ , and with  $\Theta = \{\vartheta \in \chi(\omega) \mid \ell(\vartheta) \implies\}$  it holds  $\forall \vartheta \in \Theta \ [ \beta \in \chi^+(\vartheta) ]$ .

We illustrate both propositions using our example from Fig. 4(a). For instance, transitions  $B$  and  $E$  are in strict order,  $B \rightsquigarrow E$ , as the LCA of the trivial fragments that have  $B$  and  $E$  as entries is the polygon fragment  $P2$ , cf., Fig. 4(b) and Fig. 6. Here, the *order* value for the child fragment of  $P2$  containing  $B$  is lower than the one for the child fragment that contains  $E$ , while the path from the root of the tree  $P1$  to  $P2$ , i.e.,  $\pi_{\mathcal{T}}(P1, P2)$ , does not contain any loop fragment. It holds  $D + E$  for transitions  $D$  and  $E$  due to the LCA being fragment  $B3$  in Fig. 4(b) or  $B_{\circ}1$  in Fig. 6, respectively. The fragment  $B_{\circ}1$  is a place bordered bond and, again, the path  $\pi_{\mathcal{T}}(P1, B_{\circ}1)$  does not contain any loop fragments. Transitions  $B$  and  $C$ , in turn, are an example for interleaving order,  $B \parallel C$ , as their LCA is fragment  $B2$  in Fig. 4(b). This fragment corresponds to the loop type fragment  $L1$  in Fig. 6. Derivation of the co-occurrence is illustrated using transitions  $B$  and  $C$ . We see that the path from the respective LCA (i.e.,  $B2$  in Fig. 4(b),  $L1$  in Fig. 6) to the trivial fragments having  $B$  and  $C$  as entries contains solely polygon fragments ( $P4$  and  $P5$ , respectively). However, the LCA itself is a loop fragment, such that the orientation of its child fragments  $P4$  and  $P5$  needs to be considered. There is only one child with forward orientation, namely  $P4$ . It contains transition  $B$ . Therefore, we derive  $C \gg B$ , but  $B \not\gg C$  according to Proposition 2.

Using these propositions, computation of the causal behavioural profile for a pair of transitions in a sound free-choice WF-system is very efficient.

**Corollary 2.** *The following problem can be solved in linear time.*

*Given a sound free-choice WF-system  $(N, M_i)$  and its WF-tree  $\mathcal{T}_N$ , to compute the causal behavioural profile for a pair of transitions  $(a, b)$  if  $b$  is not contained in any rigid fragment.*

*Proof.* Let  $a$  and  $b$  be two transitions and  $\beta$  be a trivial fragment of  $\mathcal{T}_N$  with  $b = \beta_{\circ}$ . Each of the behavioural relations, cf., propositions 1 and 2, requires analysis of fragments on a subpath from the root of  $\mathcal{T}_N$  to  $\beta$ . The analysis of a single fragment is performed in constant time. In the worst case, the length of the subpath is linear in size to the number of fragments in  $\mathcal{T}_N$ . The number of fragments in  $\mathcal{T}_N$  is linear to the number of flows in the WF-system [21,27,28].  $\square$

## 5.2 Computation for Rigid Fragments

Given the WF-tree, the computation of the causal behavioural profile for two transitions  $a$  and  $b$  of a WF-system as introduced above assumes that there is no rigid fragment on the path from the root of the tree to  $b$ . If  $b$  is part of a rigid fragment, derivation of the behavioural relations is more costly.

In [13], we introduced a computation of the (non-causal) behavioural profile for all transitions in  $O(n^3)$  time for sound free-choice WF-systems with  $n$  as

the number of nodes. This approach, however, has the drawback that the behavioural profile cannot be calculated for a single pair of transitions, but solely for the Cartesian product of transitions leading to increased computational complexity. For the problem of this paper, this implies computational overhead as various transitions are irrelevant for consistency analysis. Not in all cases, such irrelevant transitions might be removed in a pre-processing step without changing semantics.

While for the behavioural profile computation in polynomial time complexity is possible for sound free-choice WF-systems, the co-occurrence relation of the causal behavioural profile imposes serious challenges. In the following, we show how this relation can be derived efficiently for three subclasses, namely sound workflow T- and S-systems, and sound free-choice WF-systems that are acyclic.

First, we need an auxiliary lemma for the relation between (forwards and backwards) conflict-free paths and the co-occurrence relation. As usual, given a WF-net  $N = (P, T, F)$  a path  $\pi_N(x_1, x_k)$  is *forwards conflict-free*, iff  $x_i \in P$  implies  $|x_i \bullet| = 1$  for  $1 \leq i < k$ . The path  $\pi_N(x_1, x_k)$  is *backwards conflict-free*, iff  $x_i \in P$  implies  $|\bullet x_i| = 1$  for  $1 < i \leq k$ . The proof can be found in [6].

**Lemma 3.** *For two transitions  $x$  and  $y$  in a sound WF-system holds,*

- *if there is a forwards conflict-free path from  $x$  to  $y$ , then  $x \gg y$ .*
- *if there is a backwards conflict-free path from  $x$  to  $y$ , then  $y \gg x$ .*

The co-occurrence relation for sound workflow T-systems is derived as follows.

**Lemma 4.** *All pairs of transitions of a sound workflow T-system are in the co-occurrence relation.*

*Proof.* Let  $(N, M_i)$  be a sound workflow T-system. Let  $i \bullet = \{t_i\}$  be the initial transition (there is only one due to the structure of T-systems). For any transition  $t \in T$  any path  $\pi_N(t_i, t)$  is forwards conflict-free. Thus,  $t_i \gg t$  (Lemma 3). Consequently, all firing sequences starting with  $t_i$  imply the occurrence of every  $t \in T$ . Due to soundness, such firing sequences lead to the final marking  $M_o$ . Thus, all firing sequences  $\sigma$  with  $(N, M_i)[\sigma](N, M_o)$  contain all transitions  $t \in T$ .  $\square$

Regarding our example in Fig. 4(a), we see that Lemma 4 suffices to derive the co-occurrence relation for all pairs of transitions that can not be treated according to Proposition 2 introduced before as they are part of a rigid fragment. The subnet represented by fragment R1 in Fig. 4(b) and Fig. 6 is a T-Net, such that all transitions inside are pairwise co-occurring (e.g.,  $F \gg J$  and  $J \gg F$ ). This knowledge, in turn, is used to derive co-occurrence for pairs of transitions, in which one transition is outside the rigid. For instance, we already know  $D \gg K$ , as the trivial fragment having transition  $K$  as entry is directly contained in fragment P1 (Proposition 2 can be applied to decide co-occurrence for  $D$  and  $K$ ). As  $K$  is also the exit of the rigid fragment R1, it is co-occurring to all transitions inside R1. Thus,  $D$  is co-occurring to all these transitions, e.g.,  $D \gg H$ .

For sound workflow S-systems, the co-occurrence relation can be traced back to the notion of dominators and post-dominators known from graph theory. For

a WF-net  $N = (P, T, F)$ ,  $i$  and  $o$  as its initial and final place, and two nodes  $x, y \in X$ ,  $x$  is a dominator of  $y$ , iff for all paths  $\pi_N(i, y)$  it holds  $x \in \pi_N(i, y)$ .  $x$  is a post-dominator of  $y$ , iff for all paths  $\pi_N(y, o)$  it holds  $x \in \pi_N(y, o)$ .

**Lemma 5.** *For two transitions  $x$  and  $y$  of a sound workflow S-system holds,  $x \gg y$ , iff  $y$  is dominator or post-dominator of  $x$ .*

*Proof.* Let  $(N, M_i)$  be a sound workflow S-system and  $x, y \in T$  two transitions. In a workflow S-system, every reachable marking  $M \in [N, M_i)$  marks exactly one place, as only  $i$  is marked initially and for all transitions  $t \in T$  we know  $|\bullet t| = 1 = |t \bullet|$ . Therefore, for every firing sequence  $\sigma = t_1, \dots, t_n$  we know that there is a path  $\pi_N(t_1, t_n)$  containing all transitions of  $\sigma$  in the respective order.

$\Rightarrow$  Let  $y$  be a dominator or a post-dominator of  $x$ , assume  $x \not\gg y$ . If  $y$  is a dominator of  $x$ , then  $y \in \pi_N(i, x)$  for every path  $\pi_N(i, x)$ . Thus, any firing sequence  $\sigma$  with  $(N, M_i)[\sigma](N, M_1)$  with  $(N, M_1)[x]$  is required to contain  $y$ , i.e.,  $x \gg y$ . If  $y$  is a post-dominator of  $x$ , the argument can be turned around. for all paths  $\pi_N(x, o)$ .

$\Leftarrow$  Let  $x \gg y$  and assume that  $y$  is neither a dominator nor a post-dominator of  $x$ .  $x \gg y$  implies that any firing sequence  $\sigma$  with  $x \in \sigma$  and  $(N, M_i)[\sigma](N, M_o)$  contains  $y$  as well. Thus, all paths  $\pi_N(i, o)$  that contain  $x$  also contain  $y$ , i.e.,  $y$  is a dominator (if  $y F^+ x$ ) or post-dominator (if  $x F^+ y$ ) of  $x$ .  $\square$

For the more generic case of sound free-choice WF-systems that are acyclic, the co-occurrence relation can be traced back to the exclusiveness relation. Note that it is easy to see that two transitions that are exclusive to each other are not co-occurring. Therefore, this case is not considered in the following lemma.

**Lemma 6.** *In a sound free-choice WF-system holds, two transitions  $x$  and  $y$  that are not exclusive ( $x \not\bowtie y$ ), while  $y$  is not part of a control flow cycle ( $y \not\mathcal{P}^+ y$ ) are co-occurring, if and only if, all transitions exclusive to  $y$  are exclusive to  $x$ .*

Again, the proof of Lemma 6 is detailed in 6. Based thereon, computation of the causal behavioural profile is efficient for the respective system classes.

**Corollary 3.** *The following problem can be solved in  $O(n^3)$  time with  $n$  as the number of nodes of the system. For a sound WF-system that is a T- or S-system, or free-choice and acyclic, to compute the causal behavioural profile for a pair of transitions.*

*Proof.* Given any sound free-choice WF-system, the relations of the behavioural profile can be computed in  $O(n^3)$  time 13 (T- and S-systems are free-choice). The co-occurrence relation for the causal profile is set directly in case of a T-system (cf., Lemma 4). In case of an S-system, dominators and post-dominators are determined in linear time 30. Based thereon, co-occurrence is decided based on Lemma 5. For the case of acyclic free-choice WF-systems, co-occurrence is traced back to exclusiveness according to Lemma 6. That requires an iteration over the Cartesian product of transitions, while for each pair all other transitions are analysed, which yields a time complexity of  $O(n^3)$ . Thus, overall time complexity is  $O(n^3)$  with  $n$  as the number of nodes of the system.  $\square$

### 5.3 Implementation and Experimental Results

In order to validate our approach of deriving behavioural characteristics, we implemented the computation of the causal behavioural profiles based on WF-trees and conducted an experiment using the SAP reference model [31]. This reference model describes the functionality of the SAP R/3 system and comprises 737 EPC models. From these models, we selected those that are non-trivial (more than one element), syntactically correct, free of deadlocks or livelocks (cf., [32]), and have unambiguous instantiation semantics (cf., [33]). We also normalised multiple start and end events, and replaced OR-split and OR-join connectors with AND connectors (which does not impact on the behavioural profile, but on the causal behavioural profile). For 493 EPC models, these pre-processing steps led to a model that could be transformed into a sound free-choice WF-system following on common EPC formalisations (eg., [9]).

In our experiment, we computed the (non-causal and causal) behavioural profiles for all transitions of all 493 WF-systems separately. We grouped the models according to their size, i.e., the number of EPC nodes (the WF-systems are larger in size). Fig. 8 shows the average computation time for each model group in three experiment runs. First, we computed the behavioural profile using the approach introduced in [13] (BP-Net). Second, we derived the same profile using the WF-trees as introduced in this paper (BP-Tree). Third, we computed the causal behavioural profile (including co-occurrence) using WF-trees (CBP). Note that two WF-systems contained a rigid fragment. Both could be mapped to an S-system and, therefore, be handled as introduced in Section 5.2. To illustrate the extent to which the models of our collection suffer from the state explosion problem [34], Fig. 8 shows the average computation time for a naive creation of the reachability graph (RG). While all reachability graphs are finite (due to soundness of the WF-systems), computation takes up to tens of seconds. For all four computations, Fig. 8 depicts the polynomial (or exponential for RG) least squares regression.

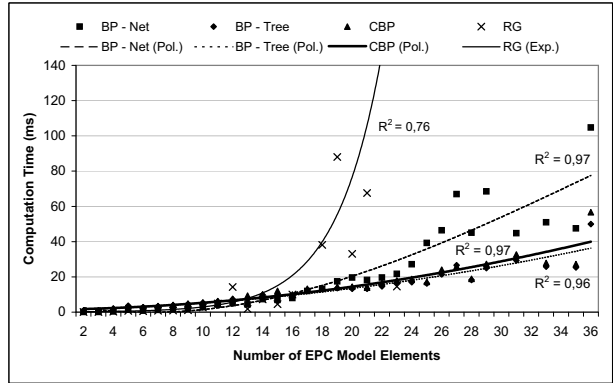


Fig. 8. Computation time relative to the size of the EPC model (Intel Core 2 Duo, 1.2 GHz, 4 GB RAM, Java 1.6)

While all reachability graphs are finite (due to soundness of the WF-systems), computation takes up to tens of seconds. For all four computations, Fig. 8 depicts the polynomial (or exponential for RG) least squares regression.

We see that the usage of WF-trees as introduced in this paper, speeds up the computation of the behavioural profile significantly compared to the existing approach. In addition, the overhead implied by our extension of the behavioural profile yielding the causal behavioural profile is negligible. Moreover,



any trace-based consistency assessment would have to explore the state space and, therefore, deal with the same computational complexity as the creation of the reachability graphs. Despite the availability of state space reduction techniques, the applicability of such an approach for real-world scenarios seems questionable.

## 6 Related Work

Clearly, our work relates to other behavioural models that have been defined for Petri nets. While we discussed causal behavioural profiles in the light of relations proposed for workflow mining [15], the well-known concurrency relation [16], and Petri net unfoldings [17,18] already in Section 3.3, their relation to common notions of behavioural equivalence deserves further explanation.

When applied in the context of model refinement and adaptation, the multitude of equivalence criteria from the linear time – branching time spectrum [35,5] has three major drawbacks. First and foremost, these notions yield a true or false answer, which has been criticised in [36]. Such notions cannot be applied to assess the amount of potential behavioural deviation. Second, it is well-known that interleaving equivalences are not invariant under *forgetful refinements* of activities [37], i.e., projection of activities. However, our initial example shows that projections are a substantial part of refining and adapting a process model towards a workflow model. These phenomena, in turn, can be quantified using the causal behavioural profile. Moreover, work on equivalence-preserving refinements for Petri nets, refer to [38] for a thorough survey, illustrates that common notions of equivalence are preserved solely under certain refinement operators. Similarly, work on net morphisms [39] and behaviour inheritance [40,41] shows that any extension of a net has to be done in a structured manner in order to preserve common equivalences. Third, notions of behavioural equivalence are computationally hard, which precludes an application for large scale industrial process models. As discussed in Section 3.3, equivalence of causal behavioural profiles is weaker than trace equivalence in order to compensate for computational efficiency.

Relations similar to those of the behavioural profile have been proposed to reason on the consistency of hardware specifications and requirements imposed by operational modules [42]. To this end, transitions of a Petri net can be classified as being sequential or parallel depending on whether there is an order between all their occurrences in all traces. In addition, these relations along with an exclusiveness relation are also defined for operations of a programming language. The authors of [42] derive these relations from the parse of an acyclic program. This, in turn, is very similar to our approach of leveraging the RPST decomposition technique. Still, the causal behavioural profile comprises further details and our approach is also applicable for cyclic nets.

The degree to which causal behavioural profiles of two related Petri nets are preserved can be used as a behavioural similarity measure. Therefore, work on causal footprints as a behavioural abstraction for determining the similarity

between processes [43] or on a trace-based similarity metric for process mining [36] is related. Further references on behavioural similarity can be found in [44].

Related work includes further applications of the tree-based decomposition for behavioural models, e.g., model transformation [20] or model abstraction [22].

## 7 Conclusions

In this paper, we addressed the problem of finding a behavioural consistency notion that is weaker than existing notions of behavioural equivalence, but can be computed efficiently. Our contribution is the definition of a causal behavioural profile that captures essential behavioural characteristics of a process. Further on, we showed the efficient computation of these profile for sound free-choice workflow systems using structural decomposition techniques under the assumption that unstructured net fragments are acyclic or can be traced back to S- or T-nets. Note that this assumption still allows the system to be cyclic, either in a structured way (bond loop fragment) or in an unstructured way (rigid fragment is a cyclic S-net). We also demonstrated the efficiency by presenting experimental results. The low polynomial complexity of our algorithms opens reasoning on behavioural consistency to industrial applications where trace based approaches do not scale.

In future research, we aim at techniques for computing causal behavioural profiles for a broader class of behavioural models, that is, systems that do not meet our assumptions on free-choiceness, soundness, and the characteristics of rigid fragments. While we addressed the suitability of our degree of consistency in a recent survey, further empirical investigations on the human perception of behavioural consistency are needed and will be tackled in future work.

## References

1. Davies, I., Green, P., Rosemann, M., Indulska, M., Gallo, S.: How do practitioners use conceptual modeling in practice? *Data Knowl. Eng.* 58(3), 358–380 (2006)
2. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of statecharts specifications. In: ICSE, pp. 54–64. IEEE CS, Los Alamitos (2007)
3. Dijkman, R., Dumas, M., García-Baueles, L., Kääriky, R.: Aligning business process models. In: EDOC. IEEE CS, Los Alamitos (2009)
4. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. *VLDB Journal* 10(4), 334–350 (2001)
5. Glabbeek, R.: The Linear Time – Branching Time Spectrum I. The semantics of concrete, sequential processes. In: *Handbook of Process Algebra*. Elsevier, Amsterdam (2001)
6. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Efficient Computation of Causal Behavioural Profiles using Structural Decomposition. Technical report 10-2010, Hasso Plattner Institute (January 2010), [http://bpt.hpi.uni-potsdam.de/pub/Public/MatthiasWeidlich/cbp\\_report.pdf](http://bpt.hpi.uni-potsdam.de/pub/Public/MatthiasWeidlich/cbp_report.pdf)

7. Aalst, W.: The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers* 8(1), 21–66 (1998)
8. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) *WS-FM 2007*. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
9. Kindler, E.: On the semantics of EPCs: A framework for resolving the vicious circle. In: Desel, J., Pernici, B., Weske, M. (eds.) *BPM 2004*. LNCS, vol. 3080, pp. 82–97. Springer, Heidelberg (2004)
10. Eshuis, R., Wieringa, R.: Tool support for verifying UML activity diagrams. *IEEE Trans. Software Eng.* 30(7), 437–447 (2004)
11. Desel, J., Esparza, J.: *Free Choice Petri Nets*. Cambridge University Press, Cambridge (1995)
12. Aalst, W.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) *ICATPN 1997*. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
13. Weidlich, M., Mendling, J., Weske, M.: Computation of behavioural profiles of process models. Technical report 08-2009, Hasso Plattner Institute (June 2009)
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Ardis, M.A., Atlee, J.M. (eds.) *FMSP*, pp. 7–15. ACM, New York (1998)
15. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. *IEEE TKDE* 16(9), 1128–1142 (2004)
16. Kovalyov, A., Esparza, J.: A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs. In: *WODES*, The Institution of Electrical Engineers, pp. 1–6 (1996)
17. McMillan, K.L.: A technique of state space search based on unfolding. *Formal Methods in System Design* 6(1), 45–65 (1995)
18. Esparza, J., Heljanko, K.: *Unfoldings: a partial-order approach to model checking*. Springer, Heidelberg (2008)
19. Weidlich, M., Weske, M., Mendling, J.: Change propagation in process models using behavioural profiles. In: *SCC*. IEEE CS, Los Alamitos (2009)
20. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) *BPM 2008*. LNCS, vol. 5240, pp. 100–115. Springer, Heidelberg (2008)
21. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. Technical Report RZ 3745, IBM (2009)
22. Polyvyanyy, A., Smirnov, S., Weske, M.: The triconnected abstraction of process models. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) *BPM 2009*. LNCS, vol. 5701, pp. 229–244. Springer, Heidelberg (2009)
23. Kiepuszewski, B., Hofstede, A., Aalst, W.: Fundamentals of control flow in workflows. *Acta Inf.* 39(3), 143–209 (2003)
24. Aalst, W.: 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)
25. Aalst, W., Hirschall, A., Verbeek, H.: An alternative way to analyze workflow graphs. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) *CAiSE 2002*. LNCS, vol. 2348, pp. 535–552. Springer, Heidelberg (2002)
26. Esparza, J., Silva, M.: Circuits, handles, bridges and nets. In: Rozenberg, G. (ed.) *APN 1990*. LNCS, vol. 483, pp. 210–242. Springer, Heidelberg (1991)
27. Gutwenger, C., Mutzel, P.: A linear time implementation of SPQR-trees. In: Marks, J. (ed.) *GD 2000*. LNCS, vol. 1984, pp. 77–90. Springer, Heidelberg (2001)

28. Battista, G.D., Tamassia, R.: On-line maintenance of triconnected components with SPQR-trees. *Algorithmica* 15(4), 302–318 (1996)
29. Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4), 541–580 (1989)
30. Alstrup, S., Harel, D., Lauridsen, P.W., Thorup, M.: Dominators in linear time. *SIAM J. Comput.* 28(6), 2117–2132 (1999)
31. Curran, T.A., Keller, G., Ladd, A.: *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Prentice-Hall, Englewood Cliffs (1997)
32. Dongen, B., Jansen-Vullers, M., Verbeek, H., Aalst, W.: Verification of the SAP reference models using EPC reduction, state space analysis, and invariants. *Computers in Industry* 58(6), 578–601 (2007)
33. Decker, G., Mendling, J.: Process instantiation. *Data Knowl. Eng.* 68 (2009)
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. Pomello, L., Rozenberg, G., Simone, C.: A survey of equivalence notions for net based systems. In: Rozenberg, G. (ed.) *APN 1992*. LNCS, vol. 609, pp. 410–472. Springer, Heidelberg (1992)
36. de Medeiros, A.K.A., Aalst, W., Weijters, A.: Quantifying process equivalence based on observed behavior. *Data Knowl. Eng.* 64(1) (2008)
37. Glabbeek, R., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. *Acta Inf.* 37(4/5), 229–327 (2001)
38. Brauer, W., Gold, R., Vogler, W.: A survey of behaviour and equivalence preserving refinements of petri nets. In: Rozenberg, G. (ed.) *APN 1990*. LNCS, vol. 483, pp. 1–46. Springer, Heidelberg (1991)
39. Winskel, G.: Petri nets, algebras, morphisms, and compositionality. *Inf. Comput.* 72(3), 197–238 (1987)
40. Basten, T., Aalst, W.: Inheritance of behavior. *JLAP* 47(2), 47–145 (2001)
41. Schrefl, M., Stumptner, M.: Behavior-consistent specialization of object life cycles. *ACM Trans. Softw. Eng. Methodol.* 11(1), 92–148 (2002)
42. Rosenblum, L., Yakovlev, A.: Analyzing Semantics of Concurrent Hardware Specifications. In: *ICPP*, vol. 3, pp. 211–218 (1989)
43. Dongen, B., Dijkman, R.M., Mendling, J.: Measuring similarity between business process models. In: Bellahsène, Z., Léonard, M. (eds.) *CAiSE 2008*. LNCS, vol. 5074, pp. 450–464. Springer, Heidelberg (2008)
44. Dumas, M., García-Bañuelos, L., Dijkman, R.M.: Similarity search of business process models. *IEEE Data Eng. Bull.* 32(3), 23–28 (2009)

# Canonical Transition Set Semantics for Petri Nets<sup>★</sup>

Yunhe Wang<sup>1,2</sup> and Li Jiao<sup>1</sup>

<sup>1</sup> State Key Laboratory of Computer Science,  
Institute of Software, Chinese Academy of Sciences, Beijing, China

<sup>2</sup> Graduate University of Chinese Academy of Sciences  
{yunhe, ljiao}@ios.ac.cn

**Abstract.** A new partial order semantics called *canonical transition set* (CTS for short) semantics is proposed for P/T nets. We first prove that it is well-defined, sound and complete, and then give a state space exploring method based on it. CTS semantics provides a compressed representation for the interleaving transition sequences of finite length. Compared with other methods, the state space exploration based on CTS semantics can avoid many redundant branches and reach all states in less steps. Furthermore, we show that CTS semantics coincides with step semantics in Foata normal form [8] for pure 1-safe nets, which gives an experimental support that CTS semantics is more efficient for state space exploration. As for a special property, deadlock, we show that all deadlocks can be detected by a method combining CTS semantics with persistent set method.

**Keywords:** Petri nets, partial order reduction, canonical transition set semantics, state space exploration, step semantics.

## 1 Introduction

Partial order method, which is used as an efficient technique for relaxing the state space explosion problem, has attracted much attention for many years. Briefly speaking, it uses the relationship between events (e.g., concurrency) in a model and either explores only the desirable executions ([15][6][17][4]) w.r.t certain property being checked or compresses some execution paths into one equivalence class (e.g., one trace [11][3]). As to the latter, the authors in [1] mapped equivalence classes of execution paths to equivalence classes of processes for finite synchronization systems, i.e., equivalent processes correspond to equivalent execution paths. For 1-safe nets particularly, an equivalence class of processes becomes a singleton, which brings great practical usage for state space exploration. Based on this, the authors in [7][8][9] improved the efficiency of property checking, such as reachability, deadlock, or some properties expressed by LTL. In [8] Heljanko used constrained Boolean circuits to verify reachability with process semantics—step executions in Foata normal form and showed good performance. For  $n$ -safe nets ( $n > 1$ ), however, it seems very difficult to find a form of execution that can represent an equivalence class of execution paths (or processes) in the sense of [1]. The

---

<sup>★</sup> This research was financially supported by the National Natural Science Foundation of China (Grants No. 60721061, No. 60633010 and No. 60970029) and the 863 program of China (Grant No. 2009 AA01Z148).

reason lies in that the individualities of tokens on the same place are distinguished and regarded as different conditions in process semantics.

In this paper we ignore such differences and tweak the equivalent occurrence sequences into one transition set sequence – the canonical transition set sequence, with the restriction that the net is finite and all occurrence sequences under consideration are finitely long. First we define a basic relation between transitions in the net in a static way, based on which transition set sequences are generated. We find that all the canonical transition set sequences have grounded mathematical properties, such as well-definedness, soundness and completeness w.r.t occurrence sequences in corresponding equivalence classes. Intuitively, the semantics combines some transitions into one step and makes them executed as early as possible. We apply this semantics in the state space exploration and find that many branching explorations can be avoided. Finally, for pure 1-safe nets, we find that all the equivalence classes of occurrence sequences based on the semantics coincide with these in [11] and the canonical transition set sequences are just the step executions in Foata norm form in [8]. As for the deadlock detection, we combine our method with persistent set method, and a comparison is drawn between canonical transition set (CTS) semantics and the CSG method [16,14].

The idea of “executing as much as possible in parallel” comes from [2] and the foata normal form of sequences (words) in [3]. In this paper, a new relation between transitions in finite Petri nets (with finite places and transitions) called order-ir is given, which concretizes the ‘indep’ one in [11,3]; our semantics is based on a framework of shift strategy, which constructs the canonical form for occurrence sequences of finite length and leads to much contribution in the exploring of state space; and furthermore, it can be found that the semantics can be combined with persistent set method for deadlock detection.

The paper is organized as follows: next in Section 2 some basic definitions for the net systems are given; then we propose the CTS semantics in Section 3 and prove its well-definedness, soundness and completeness; in Section 4 a state space exploration technique is proposed based on CTS and a deadlock detection method is given as well; after that, we discuss related work in Section 5; we finish with the conclusion in Section 6.

## 2 Basic Concepts

In this section, we introduce some basic concepts (some may be new) of Petri nets. Other definitions and properties are available in [13,12,5].

**Definition 1.** A Petri net  $\mathcal{N}$  is a 3-tuple and  $\mathcal{N} = (P, T, F)$ , where:

- $P$  is a finite set of places and  $P \neq \emptyset$ ;
- $T$  is a finite set of transitions such that  $T \neq \emptyset$  and  $P \cap T = \emptyset$ ;
- $F$  is a set of directed arcs (flow relation),  $F \subseteq (P \times T) \cup (T \times P)$ ;

Let  $x, y \in P \cup T$ , then  $\bullet x = \{y \mid (y, x) \in F\}$  and  $x^\bullet = \{y \mid (x, y) \in F\}$  are called *pre-set* and *pos-set* of  $x$ , respectively. In this paper we require that  $\bullet x \neq \emptyset \wedge x^\bullet \neq \emptyset$  for  $x \in T$  and  $\bullet x \neq \emptyset \vee x^\bullet \neq \emptyset$  for  $x \in P$ . Sometimes  $F$  is regarded as a function:  $(P \times T) \cup (T \times P) \mapsto \{0, 1\}$ .  $F(x, y) = 1$  iff  $(x, y) \in F$ . Note that we restrict our nets to those with places having unlimited capacities and arcs having only unit weights.

A *marking* of a net  $\mathcal{N}$  is a mapping  $\mathbf{M} : P \mapsto \mathbb{N}$ , where  $\mathbb{N} = \{0, 1, 2, \dots\}$  denotes the token number. A marking is always denoted as a vector, where the values of the coordinates are the token numbers in the corresponding places. A place  $p$  is *marked* by a marking  $\mathbf{M}$  if  $\mathbf{M}(p) > 0$ . A transition  $t$  is *enabled* at a marking  $\mathbf{M}$ , denoted by  $\mathbf{M}[t]$ , if each input place  $p \in {}^*t$  is marked with at least 1 token. All the transitions enabled at  $\mathbf{M}$  are denoted by a set  $\text{enabled}(\mathbf{M})$ . An enabled transition  $t$  can be chosen to *fire* and its occurrence transforms  $\mathbf{M}$  into a new marking  $\mathbf{M}'$  (symbols:  $\mathbf{M}[t]\mathbf{M}'$ , or  $\mathbf{M}t\mathbf{M}'$  for short), and for each place  $p$ :  $\mathbf{M}'(p) = \mathbf{M}(p) - F(p, t) + F(t, p)$ . Transitions may fire again at  $\mathbf{M}'$ , to generate new markings; repeat this process, then an *occurrence sequence*  $\sigma$  can be generated.

A net with an *initial marking*  $\mathbf{M}_0$  forms a system  $(\mathcal{N}, \mathbf{M}_0)$  and the occurrence sequences firing at  $\mathbf{M}_0$  constitute the system's behavior. A marking  $\mathbf{M}$  is called *reachable* from  $\mathbf{M}_0$  if there is an occurrence sequence  $\mathbf{M}_0 t_1 \mathbf{M}_1 t_2 \dots t_k \mathbf{M}_k$  and  $\mathbf{M} = \mathbf{M}_k$  for some  $k$ . All the reachable markings from  $\mathbf{M}_0$  constitute the *reachability set* of the net system. A marking  $\mathbf{M}$  is called *dead* if  $\text{enabled}(\mathbf{M}) = \emptyset$ . A net system  $(\mathcal{N}, \mathbf{M}_0)$  is called *n-safe* ( $n \geq 1$ ) if  $\mathbf{M}(p) \leq n$  holds for every place  $p$  and every reachable marking  $\mathbf{M}$ .

Throughout only finite net systems with finite reachability set are considered; the net system with an initial marking ( $\mathbf{M}_0$ ) is underlying and will not be mentioned if no confusion can arise according to the context. Note that our definition of occurrence sequence is a full version (with markings in it), in order to conform to that in [1]. Additionally, we use symbols  $p, t, \mathbf{M}, \sigma$  and the corresponding symbols with subscript to denote a place, a transition, a marking, and an occurrence sequence (or a transition sequence), respectively. Sometimes for an occurrence sequence, the markings are omitted for simplification, then we use the same symbol  $\sigma$  to denote the transition sequence. For each firing  $\mathbf{M}[t]\mathbf{M}'$  we sometimes denote it shortly by  $\mathbf{M}t\mathbf{M}'$ . The number of times a transition  $t$  happens in an transition sequence  $\sigma$  is denoted by  $\sigma(t)$ . Let  $T'$  be a multiset of transitions, and  $T'(t)$  is used to denote the number of times  $t$  appears in  $T'$ . We call transition sequence  $\sigma$  a *permutation* on  $T'$  if for each transition  $t$ :  $\sigma(t) = T'(t)$  holds. In particular, if  $T'$  is a set, then  $T'(t) = 1$  iff  $t \in T'$ .

### 3 Canonical Transition Set Semantics

In this section we will introduce the canonical transition set (CTS) semantics, which is based on the relationships between transitions in the net system.

#### 3.1 The Order-ir Relation, Order-ir Sets and Enable-Exchangeable Sets

**Definition 2.** *Two different transitions  $t_1$  and  $t_2$  are called order irrelevant to each other iff for every possible marking  $\mathbf{M}$ :  $\mathbf{M}[t_1] \wedge \mathbf{M}[t_2] \rightarrow \mathbf{M}[t_1 t_2] \wedge \mathbf{M}[t_2 t_1]$ .*

This definition determines a binary relation order-ir:  $\text{order-ir}(t_1, t_2)$  holds iff  $t_1, t_2$  are order irrelevant. Intuitively, this binary relation indicates that the occurrence of one transition does not disable another and their occurrences are commutative if they are enabled at the same marking. Note that in the definition the term 'every possible' means all the possible distributions of tokens in places, so marking  $\mathbf{M}$  is arbitrary, i.e.,  $\mathbf{M}$  can

be an unreachable marking and it is irrelevant to the initial marking, which indicates that the order-ir relation depends only on the structure of the net. This binary relation is conventionally regarded as the so-called *diamond* structure as in [10], and very similar to the *accord* with relation in [15]. For the order-ir relation, the following property is true.

**Property 1.** *In a Petri net, two different transitions  $t_1, t_2$  satisfy order – ir( $t_1, t_2$ ) iff:*

- $\bullet t_1 \cap \bullet t_2 = \emptyset$ , or else
- $\forall p \in \bullet t_1 \cap \bullet t_2 : p \in t_1^* \cap t_2^*$ .

*Proof.*  $\Leftarrow$ : If  $\bullet t_1 \cap \bullet t_2 = \emptyset$  and  $\mathbf{M}[t_1]$  and  $\mathbf{M}[t_2]$ , then let  $\mathbf{M} t_1 \mathbf{M}'$ , and for each  $p$ , we get:  $\mathbf{M}(p) \geq F(p, t_1) \wedge \mathbf{M}'(p) = \mathbf{M}(p) - F(p, t_1) + F(t_1, p) \wedge \mathbf{M}(p) \geq F(p, t_2)$ . If  $p \in \bullet t_2$  then  $p \notin \bullet t_1$ , i.e.,  $F(p, t_1) = 0$ , and then  $\mathbf{M}'(p) = \mathbf{M}(p) - F(p, t_1) + F(t_1, p) \geq \mathbf{M}(p) \geq F(p, t_2)$ , which means  $\mathbf{M}'[t_2]$  and  $\mathbf{M}[t_1 t_2]$ . Similarly  $\mathbf{M}[t_2 t_1]$  can be proved. If  $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ , let  $p \in \bullet t_1 \cap \bullet t_2$ , thus  $p \in t_1^* \cap t_2^*$ . Therefore, for each  $p \in \bullet t_2$ , it holds that  $p \notin \bullet t_1 \vee (p \in \bullet t_1 \wedge p \in t_1^*)$ , which indicates  $F(t_1, p) - F(p, t_1) \geq 0$ , i.e.,  $\mathbf{M}'(p) = \mathbf{M}(p) - F(p, t_1) + F(t_1, p) \geq \mathbf{M}(p) \geq F(p, t_2)$ . So  $\mathbf{M}'[t_2]$  and thus  $\mathbf{M}[t_1 t_2]$ . Similarly,  $\mathbf{M}[t_2 t_1]$  can be proved.

$\Rightarrow$ : for marking  $\mathbf{M}$ :  $\mathbf{M}[t_1] \wedge \mathbf{M}[t_2] \rightarrow \mathbf{M}[t_1 t_2] \wedge \mathbf{M}[t_2 t_1]$  equivalent to  $\mathbf{M}(p) \geq F(p, t_1) \wedge \mathbf{M}(p) \geq F(p, t_2) \rightarrow \mathbf{M}(p) - F(p, t_1) + F(t_1, p) \geq F(p, t_2) \wedge \mathbf{M}(p) - F(p, t_2) + F(t_2, p) \geq F(p, t_1)$  for each  $p$ . If  $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ , let  $p \in \bullet t_1 \cap \bullet t_2$ , i.e.,  $F(p, t_1) = F(p, t_2) = 1$ , then for marking  $\mathbf{M}$ :  $\mathbf{M}(p) \geq 1$  implies  $\mathbf{M}(p) - 1 + F(t_1, p) \geq 1 \wedge \mathbf{M}(p) - 1 + F(t_2, p) \geq 1$ , that is to say,  $F(t_1, p) \geq 2 - \mathbf{M}(p) \wedge F(t_2, p) \geq 2 - \mathbf{M}(p)$  holds for all such marking  $\mathbf{M}$  with  $\mathbf{M}(p) \geq 1$ . We can choose  $\mathbf{M}(p) = 1$  and get  $F(t_1, p) \geq 1 \wedge F(t_2, p) \geq 1$ , i.e.,  $p \in t_1^* \cap t_2^*$ .  $\square$

When the first condition in Property 1 is satisfied,  $t_1$  and  $t_2$  are said to be *strongly order irrelevant* (SOI for short). Note that SOI implies order-ir but not the other way round. However, for some special nets, such as pure ones (i.e., without self-loops), the order-ir relation is just the SOI relation. Note that our definition about the order-ir relation is quite different from the *concurrency* relation in [11], which is defined w. r. t. to markings, i.e., in a dynamic way. Another similar relation is the *indep* relation in trace theory. As pointed out in [3], the transitions in the indep relation act on disjoint sets of resources, intuitively; and in [11] it is described by the condition  $(\bullet t \cup \bullet t') \cap (\bullet t' \cup \bullet t) = \emptyset$  whenever  $(t, t') \in indep$  for 1-safe nets. In this sense, therefore, the order-ir relation concretizes indep for our nets.

In this paper we deal mainly with occurrence sequences with finite length. The order-ir relation induces a division for occurrence sequences with the same length.

**Definition 3.** *Let  $\sigma_1$  and  $\sigma_2$  be two occurrence sequences. Then  $(\sigma_1, \sigma_2) \in \cong$  iff  $\sigma_1 = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_i t_i \mathbf{M}_{i+1} t_{i+1} \cdots \mathbf{M}_k t_k$  and  $\sigma_2 = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_i t_{i+1} \mathbf{M}'_{i+1} t_i \cdots \mathbf{M}_k t_k$  such that  $t_i$  is order irrelevant with  $t_{i+1}$  ( $0 \leq i < k$ ).*

We define relation  $\cong^0 = id$ ;  $\cong^1 = \cong$ ;  $\cong^{i+1} = \cong^i \cdot \cong$  and  $\cong^* = \bigcup_{i \geq 0} (\cong^i)$ . Then  $\cong^*$  is an equivalence relation obviously. Each equivalence class is like a trace in the sense of Mazurkiewicz [11], but in a weaker way such that the adjacent-exchangeable transitions are in the relation of order-ir instead of independence. In comparison with the



equivalence class of occurrence sequences in [11], the definition above cares only about occurrence sequences with finite length and the order-ir relation is used in place of the concurrency relation (w.r.t. the markings).

In the next, some notions on transition sets are given.

**Definition 4.** *Transition set  $T'$  is called an order irrelevant set (OIS for short) iff  $\forall t_1, t_2 \in T' : t_1 \neq t_2 \rightarrow \text{order-ir}(t_1, t_2)$ ; an OIS is called maximal if it is not properly included by any other OIS for the same net.*

Note that any subset of an OIS is still an OIS. Because OIS's and maximal OIS's are only relevant to the structure of the net, they can be pre-computed. It is very straightforward that the computation of OIS's is equivalent to the computation of cliques in a graph, where the nodes and edges correspond to the transitions and order-ir relation of the net, respectively. In the following we suppose that all the maximal OIS's of the underlying net are  $T_1^o, T_2^o, \dots, T_s^o$ .

**Definition 5.** *Let  $M$  be a marking and  $S$  an OIS, then  $S$  is called an enabled and exchangeable transition set (EET for short) at  $M$  iff  $S \subseteq \text{enabled}(M)$ . If  $S$  is not properly included by any other EET at  $M$ , we call it maximal (MEET for short). Let  $\text{EET}(M) = \{S \mid S \text{ is an EET at } M\}$  and  $\text{MEET}(M) = \{S \mid S \text{ is a MEET at } M\}$ .*

Note that a marking  $M$  may have one or more EET's and MEET's; the empty set  $\emptyset$  is trivially an EET of any marking; when  $M$  is a dead marking, there is only one EET, i.e.,  $\text{EET}(M) = \text{MEET}(M) = \{\emptyset\}$ .  $\text{MEET}(M)$  can be determined in such a way below:

**Property 2.** *Let  $M$  be a marking, then  $\text{MEET}(M) \subseteq \{T_1^o \cap \text{enabled}(M), T_2^o \cap \text{enabled}(M), \dots, T_s^o \cap \text{enabled}(M)\}$ .*

*Proof.* Let  $S$  be a MEET at  $M$ . By Definition 5,  $S$  is an OIS and  $S \subseteq \text{enabled}(M)$ ; then by Definition 4,  $S \subseteq T_i^o$  for some  $i$  ( $1 \leq i \leq s$ ). Therefore  $S \subseteq T_i^o \cap \text{enabled}(M)$ . Assume  $(T_i^o \cap \text{enabled}(M)) - S \neq \emptyset$  and let  $t \in (T_i^o \cap \text{enabled}(M)) - S$ , then  $S \cup \{t\}$  is an EET at  $M$  by Definition 5. This leads to a contradiction since  $S$  is maximal. Therefore  $S = T_i^o \cap \text{enabled}(M)$ . Then all the MEET's at  $M$  are among these sets.  $\square$

Note that the sets  $T_1^o \cap \text{enabled}(M), T_2^o \cap \text{enabled}(M), \dots, T_s^o \cap \text{enabled}(M)$  may contain some transition sets that are EET's, but not MEET's; some empty sets or reduplicate sets may be among them as well. All such sets are redundant and can be removed for simplifying further consideration. It is clear that a MEET  $T'$  contains  $2^{|T'|} - 1$  nonempty EET's.

We show in the following that the execution of transitions in an EET at  $M$  can be in an arbitrary order.

**Property 3.** *Let  $T'$  be an EET at marking  $M$  and  $\sigma$  be a permutation on  $T'$ , then  $M[\sigma]$ .*

*Proof.* Let  $\sigma = t_0 t_1 \dots t_n$  be a permutation on  $T'$ , and  $\sigma_i = t_0 t_1 \dots t_i$  be the prefix of  $\sigma$  with length  $i + 1$  ( $0 \leq i \leq n$ ). Induction is made on  $i$ . If  $i = 0$ , it is trivially obvious that  $M[\sigma_i]$ . When  $i < n$ , assume  $M[\sigma_i]$ , i.e., there is an occurrence sequence  $M t_0 M_1 t_1 M_2 \dots t_i M_{i+1}$ . Since  $t_0$  and  $t_{i+1}$  are both enabled at  $M$  and  $\text{order-ir}(t_0, t_{i+1})$

holds, then  $\mathbf{M} [t_0 t_{i+1}]$ , i.e.,  $\mathbf{M}_1 [t_{i+1}]$ . Repeat this process and we get  $t_{i+1}$  is enabled at  $\mathbf{M}_{i+1}$ , and then  $\mathbf{M} [\sigma_{i+1}]$ . By induction hypothesis, for all  $i$  with  $0 \leq i \leq n$ :  $\mathbf{M} [\sigma_i]$ , and thus  $\mathbf{M} [\sigma]$ .  $\square$

Because of the arbitrariness of  $\sigma$  in Property 3 above, we use the transition set  $T'$  instead of  $\sigma$  to represent this occurrence and denote it by  $\mathbf{M} T'$  (or exists  $\mathbf{M}' : \mathbf{M} T' \mathbf{M}'$ ). We call such an occurrence a *weak step*. Note that it is not necessary that the transitions in  $T'$  can concurrently occur at  $\mathbf{M}$ . The following corollary is obvious by Property 3.

**Corollary 1.** *Let  $\mathbf{M} T' \mathbf{M}'$  be a weak step. If transition  $t$  is order irrelevant to all the transitions in  $T'$  and  $\mathbf{M} [t]$ , then  $\mathbf{M} [T' \cup \{t}]$ .*

### 3.2 Canonical Transition Set Sequence, Standard Shift Operation Order, and Well-Definedness

Similar to the definition of transition occurrence sequence, we can easily define the transition set sequence (TSS for short). Let  $\mathbf{M}_0 T_0 \mathbf{M}_1, \mathbf{M}_1 T_1 \mathbf{M}_2, \dots, \mathbf{M}_k T_k \mathbf{M}_{k+1}, \dots$  be all weak steps, and we get a TSS as follows:  $\mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \mathbf{M}_2 \dots \mathbf{M}_k T_k \mathbf{M}_{k+1} \dots$ . Let  $\sigma = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \dots \mathbf{M}_n t_n$  be an occurrence sequence and we can easily get a *trivial* TSS:  $\delta_0 = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \dots \mathbf{M}_n T_n$ , where  $T_i = \{t_i\}$  ( $0 \leq i \leq n$ ). There may be many TSS's for an occurrence sequence because of the order-ir relation between transitions in the sequence. In the following, we will consider the canonical TSS's with finite length.

**Definition 6.** *A TSS  $\delta = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \dots \mathbf{M}_n T_n$  is called canonical if for each  $i$  ( $1 \leq i \leq n$ ), no transition  $t \in T_i$  satisfies the following condition:*

$$\mathbf{M}_{i-1} [t] \wedge \forall t' \in T_{i-1}(\text{order} - \text{ir}(t, t')) \quad (1)$$

According to Corollary 1, the transition  $t$  satisfying (1) can be absorbed by  $T_{i-1}$ , which indicates that a shift of transitions can occur from  $T_i$  to  $T_{i-1}$ . Intuitively, a canonical TSS indicates no shift of transitions. For TSS's, the shift operations can be formalized as follows:

**Definition 7.** *Let  $\delta = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \dots \mathbf{M}_{i-1} T_{i-1} \mathbf{M}_i T_i \mathbf{M}_{i+1} \dots \mathbf{M}_n T_n$  be a TSS. A shift function  $f_i$  ( $1 \leq i \leq n$ ) is defined as  $f_i(\delta) = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \dots \mathbf{M}_{i-1} T'_{i-1} \mathbf{M}'_i T'_i \mathbf{M}_{i+1} \dots \mathbf{M}_n T_n$ , where:*

- if  $T_i = \emptyset$ ,  $f_i(\delta) = \delta$ ; otherwise
- make the assignment:  $T_{i-} := \{t | t \in T_i \text{ and } t \text{ satisfies (1)}\}$ , then  $T'_{i-1} = T_{i-1} \cup T_{i-}$  and  $T'_i = T_i - T_{i-}$ .

Intuitively, function  $f_i$  moves the transitions in  $T_{i-}$  from  $T_i$  to  $T_{i-1}$ . Note that when  $T_{i-1} = \emptyset$ , we get  $T_{i-} = T_i$  and the function actually exchanges the values of  $T_{i-1}$  and  $T_i$ , or in other words,  $\emptyset$  is moved backward. Fig. 1 is an example. A TSS can be changed by shift functions as follows (markings omitted):  $\{t_1\} \{t_3\} \{t_4\} \xrightarrow{f_1} \{t_1, t_3\} \emptyset \{t_4\} \xrightarrow{f_2} \{t_1, t_3\} \{t_4\} \emptyset$ .

**Theorem 1.** *A TSS  $\delta = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \dots \mathbf{M}_n T_n$  is canonical iff  $f_1(f_2(\dots(f_n(\delta))\dots)) = \delta$  holds.*

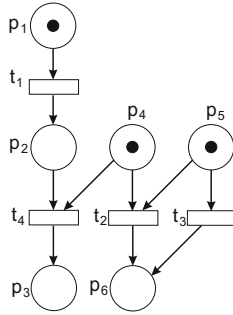


Fig. 1. A net system with initial marking  $\mathbf{M}_0 = (100110)$

*Proof.*  $\Rightarrow$ : Obviously true.

$\Leftarrow$ : If  $\delta$  is not canonical, then there is a transition  $t$  in  $T_i$  satisfying  $\square$ . Assume  $T_i$  is the most backward transition set that makes  $\square$  satisfied. Then  $f_{i+1}(f_{i+2}(\dots(f_n(\delta))\dots)) = \delta$ , and  $f_i(\delta) = \delta' \neq \delta$ , i.e., some transition  $t$  is moved from  $T_i$  to  $T_{i-1}$ . The sequence  $f_{i-1}, \dots, f_2, f_1$  after  $f_i$  can only move  $t$  and other transitions forward, and it is not reversible. Therefore,  $f_1(f_2(\dots(f_n(\delta))\dots)) = f_1(f_2(\dots(f_{i-1}(\delta'))\dots)) \neq \delta$ .  $\square$

The trivial TSS for  $\sigma = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \dots \mathbf{M}_n t_n$  is  $\delta_0 = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \dots \mathbf{M}_n T_n$ , where  $T_i = \{t_i\}$  ( $0 \leq i \leq n$ ). Theorem  $\square$  above gives an indication that shift operations  $f_n, f_{n-1}, \dots, f_2, f_1$  can be made on  $\delta_0$  many times and if a fixpoint is reached then we can get its canonical form. Because the shift operation is unidirectional and there are finitely many permutations, a fixpoint will eventually be reached. However, this shift strategy may include many redundancies. Additionally, one can choose other orders of shift operations and reach a fixpoint. We give a standard one as follows.

**Shift Strategy (SSOO).**  $\delta_0 = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \dots \mathbf{M}_n T_n$  is a trivial TSS. A standard shift operation order (SSOO for short) is given as follows:

- Step 1, make shift operations  $f_n, f_{n-1}, \dots, f_2, f_1$  in sequence;
- Step 2, make shift operations  $f_k, f_{k-1}, \dots, f_2$  in sequence, where  $k$  is the subscript of the hindmost transition set that is not empty after Step 1;
- Step 3, make shift operations  $f_{k'}, f_{k'-1}, \dots, f_3$  in sequence, where  $k'$  is the subscript of the hindmost transition set that is not empty after Step 2;
- $\dots$
- Step  $m$ , terminate the operation if the  $m$ th transition set and all the transition sets following it are empty.

We introduce some notations and symbols. After Step  $j$  or before Step  $j + 1$  the TSS is denoted by  $\delta_j$  ( $0 \leq j \leq m$ ); after the whole SSOO operation, a TSS is got, denoted by  $SSOO(\delta_0)$  or  $SSOO(\sigma)$  ( $SSOO(\delta_0) = \delta_{m-1} = \delta_m$ ). The  $i$ th transition set (marking) in  $\delta_j$  is denoted by  $T_{j,i}(\mathbf{M}_{j,i})$ . According to the SSOO operation, after Step 1, the 0th transition set will not be changed any more in the subsequent operation, i.e.,  $T_{1,0} = T_{2,0} = \dots = T_{m-1,0}$  and correspondingly  $\mathbf{M}_{1,1} = \mathbf{M}_{2,1} = \dots = \mathbf{M}_{m-1,1}$ ; generally, after Step  $i + 1$  the  $i$ th transition set will not be changed any more, i.e.,  $T_{i+1,i} = T_{i+2,i} = \dots =$

$T_{m-1,i}$  and correspondingly  $M_{i+1,i+1} = M_{i+2,i+1} = \dots = M_{m-1,i+1}$ ; we call them *stable* after the corresponding steps. The locations of transitions may change with the shift operations. In  $\sigma$  (or  $\delta_0$ ) a subscript of the transition represents its corresponding order in the sequence. Note that although two transitions  $t_i$  and  $t_j$  with different subscripts ( $i \neq j$ ) may be the same transition in the underlying net (namely  $t_i = t_j$ ), we still regard them as different appearances. We do not change the subscripts of transitions even when shift operations are made. Fig. 2 shows another net.  $\sigma = t_0 t_1 t_2 t_5 t_4$  (markings omitted here) is an occurrence sequence and consider the SSOO operation on it:

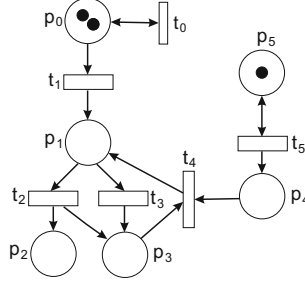


Fig. 2. A net system with initial marking  $M_0 = (200001)$

1.  $\delta_0 = M_0 \{t_0\} M_{0,1} \{t_1\} M_{0,2} \{t_1\} M_{0,3} \{t_2\} M_{0,4} \{t_5\} M_{0,5} \{t_4\}$ , where  $M_0 = (200001)$ ,  $M_{0,1} = (200001)$ ,  $M_{0,2} = (110001)$ ,  $M_{0,3} = (020001)$ ,  $M_{0,4} = (011101)$ ,  $M_{0,5} = (011111)$ ;
2. Step 1, make shift operations  $f_5, f_4, \dots, f_1$  in sequence and get  $\delta_1 = M_0 \{t_0, t_5\} M_{1,1} \{t_1\} M_{1,2} \{t_1, t_2\} M_{1,3} \emptyset M_{1,4} \emptyset M_{1,5} \{t_4\}$ , where  $M_{1,1} = (200011)$  becomes stable and  $M_{1,2} = (110011)$ ,  $M_{1,3} = (011111) = M_{1,4} = M_{1,5}$ ;
3. Step 2, make shift operations  $f_5, f_4, \dots, f_2$  in sequence and get  $\delta_2 = M_0 \{t_0, t_5\} M_{2,1} \{t_1\} M_{2,2} \{t_1, t_2\} M_{2,3} \{t_4\} M_{2,4} \emptyset M_{2,5} \emptyset$ , where  $M_{2,2} = (110011)$  becomes stable and  $M_{2,3} = (011111)$ ,  $M_{2,4} = (021001) = M_{2,5}$ ;
4. Step 3, only the shift operation  $f_3$  can be made, but no change happens because  $t_4$  is not enabled at  $M_{2,2}$ . Then  $\delta_3 = \delta_2$ ;
5. Step 4, since the 4th transition set and all the transition sets following it are empty, then terminate the operation.

Finally we get  $SSOO(\sigma) = \delta_2 = \delta_3 = \delta_4 = \{t_0, t_5\} \{t_1\} \{t_1, t_2\} \{t_4\}$ .

**Definition 8.** Let  $\delta_k$  be the TSS after Step  $k$  in SSOO. An order of the transition sets in  $\delta_k$  is defined as follows:  $T_{k,i} \ll_k T_{k,j}$  iff  $i \leq j$ . Similarly,  $t_i \ll_k t_j$  iff  $t_i \in T_{k,k_1} \wedge t_j \in T_{k,k_2} \wedge k_1 \leq k_2$ , where  $i, j$  are the subscripts (orders) of  $t_i$  and  $t_j$  in  $\sigma$  (or  $\delta_0$ ). In particular,  $t_i \ll_0 t_j$  (or  $T_i \ll_0 T_j$ ) iff  $i \leq j$ .

Intuitively,  $t_i \ll_k t_j$  means that  $t_i$  appears *not later* than  $t_j$  in  $\delta_k$ . From the definition we can conclude that  $t_i \ll_k t_j \wedge t_j \ll_k t_i$  implies that  $t_i$  and  $t_j$  are in the same transition set in  $\delta_k$ . Then the definition of the order can be easily extended between a transition and a set:  $T_{k,k_1} \ll_k t_j$  iff  $t_j \in T_{k,k_2} \wedge k_1 \leq k_2$ .

**Corollary 2.** Suppose  $\delta_0 = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_n T_n$  is a trivial TSS. Let  $T_0^c = \{t_j | t_j \in \text{enabled}(\mathbf{M}_0) \wedge \forall t_i (t_i \ll_0 t_j \wedge i \neq j \rightarrow \text{order} - \text{ir}(t_i, t_j))\}$ . Then  $T_{1,0} = T_{2,0} = \cdots = T_{m-1,0} = T_0^c$  (w. r. t. SSOO).

*Proof.* Let  $t_j \in T_0^c$ . If  $j = 0$ , it trivially holds that  $t_j \in T_{1,0}$ . When  $j > 0$  we get  $t_0 \ll_0 t_j$  and  $\text{order} - \text{ir}(t_0, t_j)$ . Since  $t_j \in \text{enabled}(\mathbf{M}_0)$ , then  $t_j \in \text{enabled}(\mathbf{M}_1)$  holds according to Corollary 1. For the same reason, we can deduce that  $t_j \in \text{enabled}(\mathbf{M}_i)$  for all  $i (i \leq j)$ . Then the succession of shift operations  $f_j, f_{j-1}, \dots, f_2, f_1$  in Step 1 (in SSOO) makes  $t_j$  passe through  $T_{j-1}, T_{j-2}, \dots, T_1, T_0$  in sequence, i.e.,  $t_j \in T_{1,0}$ . Conversely, if  $t_j \in T_{1,0}$  and  $j > 0$ , then  $t_j \in \text{enabled}(\mathbf{M}_0)$  (otherwise it cannot be moved to  $T_0$ ). The serial movements of  $t_j$  from  $T_j$  to  $T_0$  indicate that  $t_j$  is order irrelevant to all the transitions that appear not later than it in  $\delta_0$  according to condition (II), i.e.,  $t_i \ll_0 t_j$  implies  $\text{order} - \text{ir}(t_i, t_j)$ .  $\square$

After Step 1 in SSOO, we get  $\delta_1 = \mathbf{M}_{1,0} T_{1,0} \mathbf{M}_{1,1} T_{1,1} \cdots \mathbf{M}_{1,k_1} T_{1,k_1}$ , and similarly, we construct  $T_1^c = \{t_j | t_j \in \text{enabled}(\mathbf{M}_{1,1}) \wedge \forall t_i (T_{1,1} \ll_1 t_i \ll_1 t_j \wedge i \neq j \rightarrow \text{order} - \text{ir}(t_i, t_j))\}$ . For the same reason,  $T_{2,1} = T_{3,1} = \cdots = T_{m-1,1} = T_1^c$ . Similarly,  $T_k^c = \{t_j | t_j \in \text{enabled}(\mathbf{M}_{k,k}) \wedge \forall t_i (T_{k,k} \ll_k t_i \ll_k t_j \wedge i \neq j \rightarrow \text{order} - \text{ir}(t_i, t_j))\}$  is constructed for all steps and then  $T_{k+1,k} = T_{k+2,k} = \cdots = T_{m-1,k} = T_k^c$ . Note that after Step  $m-1$ , there is no nonempty transition set following  $T_{m-1,m-1}$ , and  $T_{m-1}^c$  is defined to be equal to  $T_{m-1,m-1}$ . Therefore, the SSOO operation actually provides a procedure to iteratively compute  $T_k^c$ , i.e., Step  $k+1$  computes  $T_k^c (k \geq 0)$ , and then  $SSOO(\delta_0) = \mathbf{M}_{m-1,0} T_{m-1,0} \mathbf{M}_{m-1,1} T_{m-1,1} \cdots \mathbf{M}_{m-1,m-1} T_{m-1,m-1} = \mathbf{M}_0 T_0^c \mathbf{M}_{1,1} T_1^c \cdots \mathbf{M}_{m-1,m-1} T_{m-1}^c$ .

**Lemma 1.** Let  $T_k^c (0 \leq k \leq m-1)$  be the transition sets constructed by the SSOO operation. For each  $k > 0$ , there is no transition in  $T_k^c$  that makes (IV) satisfied, therefore  $SSOO(\delta_0)$  is a canonical TSS.

*Proof.* Suppose that there is a transition  $t_s$  in  $T_k^c$  satisfying (IV), then  $t_s$  is not absorbed by  $T_{k-1}^c$  in Step  $k$  but absorbed by  $T_k^c$  in Step  $k+1$ , and  $t_s$  satisfies condition (II) which implies  $t_s \in \text{enabled}(\mathbf{M}_{k-1,k-1})$ . Then there exists a transition  $t_r$  such that  $T_{k-1,k-1} \ll_{k-1} t_r \ll_{k-1} t_s \wedge r \neq s$  holds but  $\text{order} - \text{ir}(t_s, t_r)$  not, according to the definition of  $T_{k-1}^c$ . Note that in all the following shifts in SSOO,  $t_s$  cannot pass through  $t_r$  to move forward. The fact that Step  $k+1$  can make  $t_s$  absorbed by  $T_k^c$  indicates that  $t_r$  is in  $T_{k-1}^c$  after Step  $k$ , which gives a contradiction that  $t_s$  can be moved into  $T_{k-1}^c$ . Therefore, there is no transition that can move forward in  $SSOO(\delta_0)$  and then,  $SSOO(\delta_0)$  is a canonical TSS.  $\square$

**Lemma 2.**  $SSOO(\delta_0)$  ( $SSOO(\sigma)$ ) is the unique canonical TSS for  $\sigma$ .

*Proof.*  $SSOO(\delta_0) = \mathbf{M}_0 T_0^c \mathbf{M}_{1,1} T_1^c \cdots \mathbf{M}_{m-1,m-1} T_{m-1}^c$  is the canonical TSS constructed by SSOO and we assume that  $\delta' = \mathbf{M}_0 T_0^{c'} \mathbf{M}_1' T_1^{c'} \cdots \mathbf{M}_k T_k^{c'}$  is another canonical TSS for  $\sigma$ , which is constructed by shift operations of another order. We will prove  $\delta' = SSOO(\delta_0)$ . Firstly we prove  $T_0^c = T_0^{c'}$ . For each transition  $t_j \in T_0^c = \{t_j | t_j \in \text{enabled}(\mathbf{M}_0) \wedge \forall t_i (t_i \ll_0 t_j \wedge i \neq j \rightarrow \text{order} - \text{ir}(t_i, t_j))\}$ , note a fact that  $t_j$  is order irrelevant with all the transitions appearing not later than it in all the intermediate TSS's in the construction of  $\delta'$ : in  $\delta_0$  it is obvious; afterwards, all the transitions that jump over  $t_j$  in the process are order irrelevant with  $t_j$  according to (II). In other words,  $t_j$

keeps this property, and in the canonical  $\delta'$  it cannot shift, thus  $t_j \in T_0^{c'}$ . On the other hand, if transition  $t_j$  is not in  $T_0^c$ , then it is not enabled at  $\mathbf{M}_0$  or there exists a transition  $t_i$  that makes  $t_i \ll_0 t_j \wedge i \neq j$  hold but  $order - ir(t_i, t_j)$  not. The former case implies  $t_j \notin T_0^{c'}$  and the latter case indicates that  $t_j$  will never jump over  $t_i$  in the process of shift operation, which also means  $t_j$  cannot be in  $T_0^{c'}$ . Then  $T_0^c = T_0^{c'}$  and  $\mathbf{M}'_1 = \mathbf{M}_{1,1}$  as well. The suffixes attained from  $SSOO(\delta_0)$  and  $\delta'$  by deleting the heads  $\mathbf{M}_0 T_0^c$  and  $\mathbf{M}_0 T_0^{c'}$  (respectively) are also two canonical TSS's, but for the occurrence sequence attained by deleting the transitions in  $T_0^c$  ( $T_0^{c'}$ ) from  $\sigma$ . Then by induction hypothesis, we get  $\delta' = SSOO(\delta_0)$ .  $\square$

The two lemmas above show that for each occurrence sequence, its canonical TSS is unique and it is irrelevant to the shift operation order. This determines a function as follows.

**Theorem 2 (well-definedness).** *There exists a canonical function  $\varphi$  that maps each occurrence sequence  $\sigma$  of finite length to its canonical TSS, i.e.,  $\varphi(\sigma) = SSOO(\sigma)$ .*

### 3.3 Soundness and Completeness

Let  $\bar{\delta} = \mathbf{M}_0 T_0^c \mathbf{M}'_1 T_1^c \cdots \mathbf{M}'_k T_k^c$  be a canonical TSS and  $\sigma_0, \sigma_1, \dots, \sigma_k$  are permutations on  $T_0^c, T_1^c, \dots, T_k^c$ , respectively. According to Property [B1](#)  $\sigma = \sigma_0 \sigma_1 \cdots \sigma_k$  is an occurrence sequence (called the *permutations* on  $\bar{\delta}$ ) and it is obvious that  $\varphi(\sigma) = \bar{\delta}$ . However, there are some occurrence sequences  $\sigma'$  that cannot be constructed in such a way but also have  $\varphi(\sigma') = \bar{\delta}$ . For example, consider the occurrence sequence  $\sigma = t_0 t_1 t_2 t_5 t_4$  in Fig. [2](#) with its canonical TSS  $SSOO(\sigma) = \{t_0, t_5\} \{t_1, t_2\} \{t_4\}$ . Note that transition  $t_5$  jumps over the middle two sets to  $T_0$  and then makes  $\sigma$  not be any permutation on  $SSOO(\sigma)$ .

In the following we determine when two occurrence sequences have the same canonical TSS. The theorem below shows that the canonical function  $\varphi$  is sound and complete with respect to  $\cong^*$ .

**Theorem 3 (soundness and completeness).** *Let  $\sigma$  and  $\sigma'$  be two occurrence sequences with the same finite length, then  $\varphi(\sigma) = \varphi(\sigma')$  iff  $(\sigma, \sigma') \in \cong^*$ .*

*Proof.*  $\Leftarrow$ (completeness): Assume  $\sigma \cong^i \sigma'$ , i.e., there exist occurrence sequences  $\sigma_1, \sigma_2, \dots, \sigma_i$  such that  $\sigma \cong \sigma_1, \sigma_1 \cong \sigma_2, \dots, \sigma_{i-1} \cong \sigma_i$  and  $\sigma_i = \sigma'$ . When  $i = 0$ , we get  $\sigma = \sigma'$  and  $\varphi(\sigma) = \varphi(\sigma')$  trivially. If  $i \geq 1$ , then for  $\sigma \cong \sigma_1$ , by the definition of  $\cong$ , we have  $\sigma = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_j t_j \mathbf{M}_{j+1} t_{j+1} \cdots \mathbf{M}_k t_k, \sigma_1 = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_j t_{j+1} \mathbf{M}'_{j+1} t_j \cdots \mathbf{M}_k t_k$  such that  $t_j$  is order irrelevant with  $t_{j+1}$ . We adopt such a shift operation order that shift function  $f_{j+1}$  is applied first for both trivial TSS's. Because transitions in  $T_j$  and  $T_{j+1}$  in the trivial TSS's are both enabled at  $M_j$  and order irrelevant to each other, we get the same resultant TSS after shift operation  $f_{j+1}$ , which implies  $\varphi(\sigma) = \varphi(\sigma_1)$ . Then by transitivity, we get  $\varphi(\sigma) = \varphi(\sigma_1) = \varphi(\sigma_2) = \cdots = \varphi(\sigma_i) = \varphi(\sigma')$ .

$\Rightarrow$ (soundness): Let  $\delta_s$  be an arbitrary TSS for  $\sigma$  and  $f_i$  ( $i > 0$ ) a shift function on  $\delta_s$ . First we prove that the permutations on  $\delta_s$  are all  $\cong^*$ -equivalent to those on  $f_i(\delta_s)$ . It is apparent that permutations on the same TSS are  $\cong^*$ -equivalent to each other according to the definition of  $\cong$ . By the definition of  $f_i$ , we have  $\delta_s = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_{i-1} T_{i-1} \mathbf{M}_i T_i$

$M_{i+1} \cdots M_k T_k$ ,  $f_i(\delta_s) = M_0 T_0 M_1 T_1 \cdots M_{i-1} T'_{i-1} M'_i T'_i M_{i+1} \cdots M_k T_k$  such that  $T_{i-} = \{t \mid t \in T_i \text{ and } t \text{ satisfies } \textcircled{D}\}$ ,  $T'_{i-1} = T_{i-1} \cup T_{i-}$  and  $T'_i = T_i - T_{i-}$  (Note that it can be concluded trivially for  $T_i = \emptyset$ ). Then it is also apparent that the permutations on both  $\delta_s$  and  $f_i(\delta_s)$  are all  $\cong^*$ -equivalent to permutations on TSS  $\delta_- = M_0 T_0 M_1 T_1 \cdots M_{i-1} T_{i-1} M_i T_{i-} M T'_i M_{i+1} \cdots M_k T_k$ , then they are  $\cong^*$ -equivalent to each other by transitivity. Therefore, the permutations on all the TSS's in the process of constructing  $\varphi(\sigma)$  are  $\cong^*$ -equivalent by transitivity of  $\cong^*$ , which indicates that  $\sigma$  and  $\sigma'$  are both  $\cong^*$ -equivalent to permutations on  $\varphi(\sigma)$  ( $\varphi(\sigma')$ ), and then they are  $\cong^*$ -equivalent to each other as well.  $\square$

Theorem 3 indicates that the canonical function  $\varphi$  actually determines an equivalence partition for all the occurrence sequences with the same finite length: occurrence sequences are in the same equivalence class iff they are  $\cong^*$ -equivalent to each other. We use  $\varphi^{-1}(\bar{\delta})$  to denote such an equivalence class for each canonical TSS  $\bar{\delta}$ .

## 4 State Space Exploration Based on CTS

Intuitively, CTS always tries to execute the transitions as early as possible. If a transition chosen to fire at a marking is exchangeable with (i.e., absorbed by, w.r.t order-ir) the transitions that have fired to reach that marking, then it can be inferred that this transition has been chosen by another canonical TSS. For example in Fig. 1, a canonical TSS is  $M_0 \{t_1, t_3\} M_1$  (where  $M_1 = (010101)$ ) and at  $M_1$  transition  $t_2$  is enabled. However,  $t_2$  is exchangeable with  $t_1$  and  $t_3$  (w.r.t order-ir) and cannot be chosen to fire at  $M_1$  in this TSS. There is another canonical TSS in which  $t_2$  is chosen to fire, i.e.,  $M_0 \{t_1, t_3, t_2\} M$  (where  $M = (010002)$ ). In other words, in order to explore the state space based on CTS, we should choose transitions that cannot be absorbed by the transition set that precedes it in the TSS. In the example, only  $t_4$  is chosen at  $M_1$  to generate a new canonical TSS:  $M_0 \{t_1, t_3\} M_1 \{t_4\} M_2$  (where  $M_2 = (001001)$ ). Essentially, CTS compresses some transitions into one step and gives up some branches to be explored. In the following we first compute the canonical TSS's in a dynamic perspective to achieve a complete state exploring, and then combine CTS semantics with the persistent set method to detect deadlock in a more compressed way.

### 4.1 Complete State Exploring Based on CTS

**Definition 9.** Let  $M_0 T_0 M_1 T_1 \cdots M_k T_k M$  be a canonical TSS. Transition  $t$  is called under-chosen (UC for short) at  $M$  if  $t \in \text{enabled}(M)$  and  $T_k \cup \{t\}$  is not an EET at  $M_k$  (i.e.,  $t$  cannot be absorbed by  $T_k$ ). An EET  $T'$  at  $M$  is called under-chosen if each transition in  $T'$  is UC. An UC set  $T'$  is called maximal (MUC for short) if it is not properly included by any other UC set at that marking.

The UC sets are closely related to canonical TSS's.

**Theorem 4.** A TSS  $M_0 T_0 M_1 T_1 \cdots M_k T_k M$  is canonical iff for all  $i$  ( $0 \leq i \leq k$ )  $T_i$  is an UC set at  $M_i$ .

*Proof.* It can be easily concluded by the definitions of canonical TSS and UC sets.  $\square$

Note that in Definition 9, all the transitions that can be absorbed by  $T_k$  can be easily determined. We denote such a transition set associated with  $\mathbf{M}$  by  $\text{sleep\_set}(\mathbf{M})$ <sup>1</sup>, and  $\text{sleep\_set}(\mathbf{M}) = \{t \mid t \in \text{enabled}(\mathbf{M}_k) \wedge \forall t' \in T_k(\text{order} - \text{ir}(t, t'))\}$ . Then for a canonical TSS  $\mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_k T_k \mathbf{M}$  all the UC transitions at  $\mathbf{M}$  just constitute the set  $\text{enabled}(\mathbf{M}) - \text{sleep\_set}(\mathbf{M})$ .

**Theorem 5.** *Let  $T_1^o, T_2^o, \dots, T_s^o$  be all the maximal OIS's of the underlying net. A TSS  $\mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_k T_k \mathbf{M}$  can be constructed iteratively as follows for all  $i$  ( $0 \leq i \leq k$ ):*

- if  $i = 0$ , then choose  $T_0$  as one EET at  $\mathbf{M}_0$ ;
- if  $i > 0$  and  $T_{i-1}$  is an UC set at  $\mathbf{M}_{i-1}$ , then let  $T_c = \text{enabled}(\mathbf{M}_i) - \text{sleep\_set}(\mathbf{M}_i)$  and get the sets  $T_1^o \cap T_c, T_2^o \cap T_c, \dots, T_s^o \cap T_c$ . Choose  $T_i$  such that  $T_i \subseteq T_j^o \cap T_c$  for some  $j$  ( $1 \leq j \leq s$ ).

Then for all  $i$  ( $0 \leq i \leq k$ )  $T_i$  is an UC set at  $\mathbf{M}_i$  and therefore  $\mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_k T_k \mathbf{M}$  is a canonical TSS.

*Proof.* It can be proved by induction on  $i$ . When  $i = 0$ ,  $T_0$  is an EET at  $\mathbf{M}_0$ , which is also an UC set since no transition set precedes it. When  $i > 0$ , assume  $T_{i-1}$  is an UC set at  $\mathbf{M}_{i-1}$ , then  $T_c = \text{enabled}(\mathbf{M}_i) - \text{sleep\_set}(\mathbf{M}_i)$  includes exactly all the UC transitions at  $\mathbf{M}_i$ . However,  $T_c$  is not necessarily an UC set because the transitions in it may be not order irrelevant with each other. When an intersection is made between  $T_c$  and the maximal IOS's, we get all the MUC sets among the sets  $T_1^o \cap T_c, T_2^o \cap T_c, \dots, T_s^o \cap T_c$ , similar to Property 2. Some sets among  $T_1^o \cap T_c, T_2^o \cap T_c, \dots, T_s^o \cap T_c$  may be not maximal, but they are still UC sets.  $T_i$  is chosen as a subset of them, then  $T_i$  is an UC set at  $\mathbf{M}_i$ . Therefore  $\mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_k T_k \mathbf{M}$  is a canonical TSS.  $\square$

Since the sets  $T_1^o \cap T_c, T_2^o \cap T_c, \dots, T_s^o \cap T_c$  cover all the possible UC sets at  $\mathbf{M}_i$ , then we have the following corollaries.

**Corollary 3.** *In Theorem 5 the construction strategy for canonical TSS  $\mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_k T_k \mathbf{M}$  explores all the possible canonical TSS's within length  $k+1$  in the state space.*

**Corollary 4.** *If the state space is finite and the transition sets  $T_i$  is chosen by the strategy in Theorem 5 then there exists a bound  $k$  such that all the canonical TSS's of length  $k$  explore all the reachable markings in the state space.*

*Proof.* Assume  $\mathbf{M}'$  is a reachable marking, then there exists an occurrence sequence  $\sigma$  of finite length such that  $\mathbf{M}_0 \sigma \mathbf{M}'$  and all the markings in  $\sigma$  appear only once (this sequence can be always acquired by deleting the subsequences between any two appearances of the same markings). Make SSOO operation on  $\mathbf{M}_0 \sigma \mathbf{M}'$  and we can get a canonical TSS  $\delta = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_i T_i \mathbf{M}'$ . According to Corollary 3,  $\delta$  is explored by length  $i+1$ , which cannot exceed the length of  $\sigma$ . As to all the reachable markings, let  $k$  be the maximal value of the lengths of such  $\delta$ 's. Then we get the conclusion.  $\square$

Note that  $k$  cannot exceed the number of states in the space in the worst case. Since all the reachable markings in the state space are explored by such a strategy, we call

<sup>1</sup> This notion is borrowed from [6], where sleep set is used as a transition set to be excluded in the exploring. We use it in a similar way.



it a *complete-state* exploring. Note that by the strategy in Theorem 5, when a marking is visited more than once (i.e., reached from the initial marking by different canonical TSS's), all the possible next UC sets are explored. This can preserve all the possible canonical paths (of the bounded length) from the initial marking.

The state space exploration based on CTS is different from that traditional one based on interleaving semantics since it tries to make transitions fire as early as possible: many transitions fire at one step as a transition set. Therefore, the CTS exploration can reduce the number of steps besides avoiding the exploration of some redundant branches. The drawback is obvious as well. The complexity of CTS lies in the computation of UC sets at each reachable marking and there are often exponentially many UC sets w. r. t. the number of UC transitions at each marking. Below is a comparison between the CTS exploration and the traditional one for the net in Fig. 1. In Fig. 3 markings are represented by solid dots and the numbers on arcs are the indexes of the corresponding transitions. The two graphs are the trails of the two explorations, respectively. The CTS exploring reaches all the states within bound 2 (see the right figure in Fig. 3). Note that the traditional exploring exhibits a *lattice* shape while the CTS exploring exhibits a *tree* shape in the drawing. The net has two maximal OIS's:  $\{t_1, t_2, t_3\}, \{t_1, t_3, t_4\}$  and the initial marking is  $M_0 = (100110)$  and  $enabled(M_0) = \{t_1, t_2, t_3\}$ . The intersections between  $enabled(M_0)$  and the sets in maximal OIS's are  $\{t_1, t_2, t_3\}$  and  $\{t_1, t_3\}$ . Then we get 7 EET's at  $M_0$ . Assume  $\{t_1\}$  is chosen to fire (i.e.,  $T_0 = \{t_1\}$ ) so that  $M_0 \{t_1\} M_1$ . Then  $enabled(M_1) = \{t_4, t_2, t_3\}$ . Note that  $t_2, t_3$  can be absorbed by  $T_0$ , then only  $t_4$  is an UC transition and  $T_1 = \{t_4\}$ . Afterwards we get a sequence  $M_0 \{t_1\} M_1 \{t_4\} M_2$ , where  $M_2 = (001010)$ . Similarly,  $enabled(M_2) = \{t_3\}$  and  $t_3$  can be absorbed by  $T_1$ , so there is no transition that can be chosen to fire at  $M_2$ . Note that if there is no outgoing arc from a vertex in the CTS exploring graph, it does not necessarily indicates that the corresponding marking is dead. Fig. 3 illustrates that the two explorations both reach all the reachable markings but the CTS one is more efficient since it combines several transitions into one step and avoids some redundant branches.

The CTS semantics travels less edges to reach all the states and the number of edges is at least the number of states minus 1 when each state is visited only once. It is often the case that some states are visited only once because the redundant branches are not explored to reach these states one more time, such as the exploration in Fig. 3. This situation depends on how many transitions can occur together. In the worst case, when

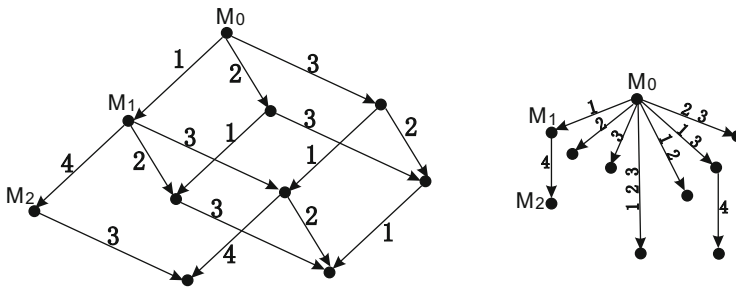


Fig. 3. The traditional state space exploring strategy and the one based on CTS for net in Fig. 1

no two transitions can occur together, the CTS exploring degenerates into the traditional one. Note that if the graph (trails) for the CTS exploring is completely constructed (i.e., the bound  $k$  is big enough to explore all the states), then not all the paths in the graph are canonical since one node can be visited more than once. Besides, it makes little sense to construct the graph since all the states are preserved and only some edges are abandoned. However, if a specific property (e.g., deadlock) is considered, some states (and more branches) will be abandoned to make the exploration more efficient, as this will be demonstrated in the next subsection.

## 4.2 Deadlock Detection Based on CTS

First we introduce the notion of persistent set [174] and show how it can be combined with CTS to detect deadlocks.

**Definition 10.** A nonempty set  $P_M$  associated with a marking  $M$  is persistent iff:

- $P_M \subseteq \text{enabled}(M)$ , and
- for any occurrence sequence  $\sigma = M t_0 M_1 t_1 \cdots M_n t_n$  ( $n \geq 0$ ) starting from  $M$ , if  $\{t_0, t_1, \dots, t_n\} \cap P_M = \emptyset$ , then for  $\forall t \in \{t_0, t_1, \dots, t_n\}$ ,  $\forall t' \in P_M$ :  $\text{order} - \text{ir}(t, t')$ .

In other words, the occurrences of transitions not in  $P_M$  cannot change the enabledness of transitions in  $P_M$ ; transitions in  $P_M$  keep their enabledness from  $M$  until some transition in  $P_M$  is chosen to fire. We denote the set of all the persistent sets associated with marking  $M$  by  $PS(M)$ , i.e.,  $PS(M) = \{P_M \mid P_M \text{ is a persistent set associated with } M\}$ . Note that  $\emptyset$  is not regarded as a persistent set; if  $M$  is not a dead marking, then  $\text{enabled}(M)$  is trivially a persistent set; if  $T_1, T_2 \in PS(M)$  then  $T_1 \cup T_2 \in PS(M)$ . The persistent set has the property of transitivity:

**Property 4.** Let  $P_M \in PS(M)$ ,  $T'$  be an EET at  $M$  such that  $M T' M'$ . If  $T' \cap P_M = \emptyset$  then  $P_M \in PS(M')$ .

*Proof.* It can be easily proved by Definition 10. □

**Theorem 6.** Let  $\sigma$  be an occurrence sequence such that  $M_0 \sigma D$ , where  $D$  is a dead marking.  $\varphi(\sigma) = M_0 T_0 M_1 T_1 \cdots M_{i-1} T_{i-1} M_i T_i M_{i+1} \cdots M_n T_n D$  is the canonical TSS for  $\sigma$ . Then for all  $i$  ( $0 \leq i \leq n$ ) and  $\forall P_{M_i} \in PS(M_i)$ :  $T_i \cap P_{M_i} \neq \emptyset$ .

*Proof.* Assume there is an  $i$  ( $0 \leq i \leq n$ ) and a persistent set  $P_{M_i}$  associated with  $M_i$  such that  $T_i \cap P_{M_i} = \emptyset$ . If  $i = n$ , then by Property 4,  $P_{M_n}$  is also a persistent set associated with  $D$ , contradicting that  $D$  is a dead marking. Then there is a  $j$  such that  $i < j \leq n$  and  $T_j \cap P_{M_i} \neq \emptyset$ ; otherwise, the same contraction can be induced by the transitivity of persistent set. Let  $T_j$  be the first transition set after  $T_i$  such that  $T_j \cap P_{M_i} \neq \emptyset$  holds, and let  $t \in T_j \cap P_{M_i}$ . Since  $P_{M_i}$  is persistent at  $M_{j-1}$  and  $T_{j-1} \cap P_{M_i} = \emptyset$ , then  $t$  is order irrelevant to all the transitions in  $T_{j-1}$  by Definition 10. Therefore,  $t$  can be moved from  $T_j$  to  $T_{j-1}$ . This leads to a contradiction since  $\varphi(\sigma)$  is a canonical TSS. □

Theorem 6 indicates that to preserve all the dead markings and the corresponding sequences, it is not necessary to explore all the UC sets  $T_i$  at marking  $M_i$ : if  $T_i \cap P_{M_i} = \emptyset$

for some  $P_{M_i} \in PS(M_i)$ , then the exploring of  $T_i$  can be ignored. This method avoids many branches to be explored. In the following, we use  $NS(PS(M_i))$  to denote the set of all the next necessary steps at  $M_i$ , i.e., all the  $T_i$ 's in Theorem 6. In particular, if there are some persistent sets that are singletons associated with marking  $M_i$ , then all the steps in  $NS(PS(M_i))$  should include all the transitions in these singletons. Also note that if there exists a  $P_{M_i} \in PS(M_i)$  such that  $P_{M_i} \subseteq sleep\_set(M_i)$ , then it follows that  $NS(PS(M_i)) = \emptyset$ , which indicates that  $M_i$  can never reach a deadlock by a canonical path and  $M_i$  itself is a 'fake' deadlock.

Then when the bound  $k$  (by Corollary 4) is big enough, all deadlocks can be detected. However, it is very complicated and not practical to compute  $PS(M_i)$ . We hope to obtain a set that is easy to compute and at the same time can eliminate as many branches as possible. First note that for any marking  $M$ , if  $T_1, T_2 \in PS(M)$  and  $T_1 \subseteq T_2$ , then  $T_2$  is redundant in the detection of deadlocks by Theorem 6. All such redundant persistent sets can be eliminated from  $PS(M)$ . Additionally, we have a tradeoff:

**Corollary 5.**  $S \subseteq PS(M)$  implies  $NS(PS(M_i)) \subseteq NS(S)$ .

*Proof.* By Theorem 6, all the steps  $T'$  in  $NS(PS(M_i))$  satisfies the condition:  $T' \cap P_M \neq \emptyset$  for each  $P_M \in PS(M)$ . Since  $S$  is a subset of  $PS(M)$ , the same condition is satisfied with respect to  $S$ , and then  $T' \in NS(S)$ . This completes the proof.  $\square$

At each reachable marking  $M$ , if  $S$  is used in place of  $PS(M)$ , then more branches will be explored and of course all the deadlocks are preserved (assume the bound  $k$  is big enough). In particular, if  $S = \emptyset$ , then it is just the complete state exploring and achieves no more reduction for the exploring. Here is a tradeoff between the computation of persistent sets and the reduction of exploration. We recommend a subset of  $PS(M)$  for each marking  $M$ , in which each persistent set is induced by an enabled transition. We denote it by  $PS S(M)$  (the second 'S' means 'subset') and it has at most as many elements as  $enabled(M)$ . More details on how to induce a persistent set by an enabled transition and more methods to compute them can be referred to in [4]. There may be many alternatives for  $PS S(M)$ : some transition in  $enabled(M)$  can induce more than one persistent set and different choices lead to different variants of  $PS S(M)$ .  $PS S(M)$  is recommended mainly for two reasons:

1. There are some ready-made algorithms on how to induce a persistent set from an enabled transition, such as those in [4];
2. It is often the case that  $PS S(M)$  is very close or even equal to  $PS(M)$  and achieves great reductions in the exploring.

A Graph (i.e., the trails of the exploring for deadlocks) for the net in Fig. 1 within bound  $k = 2$  is constructed based on  $PS S(M)$ , as shown in Fig. 4. At the initial marking  $M_0$ ,  $PS S(M_0) = \{\{t_1\}, \{t_1, t_2\}, \{t_3\}\}$ , in which the three persistent sets are induced by  $t_1, t_2, t_3$ , respectively. Note that  $\{t_1, t_2\}$  is redundant. Then only UC sets  $\{t_1, t_2, t_3\}$  and  $\{t_1, t_3\}$  are chosen to fire (i.e.,  $NS(PS S(M_0)) = \{\{t_1, t_3\}, \{t_1, t_2, t_3\}\}$ ). Finally, the graph becomes very small and two deadlocks are detected within two steps.

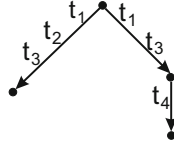


Fig. 4. The  $PS S(M)$  exploration for the net in Fig. 1

## 5 Related Work and Experimental Support for CTS

In this section, we introduce three related work: the first two from [1] and [8], where some equivalent results are given for pure 1-safe nets; the second from [16], where a covering step graph (CSG for short) is constructed for the state space reduction. We prove for the former work that the equivalence classes of occurrence sequences coincide with those classes in [1], and the canonical TSS’s coincide with the foata normal form of step executions in [8], for pure 1-safe nets. Then an experimental support is got for CTS from [8]. Finally, a comparison is drawn between CSG and CTS when both are combined with the persistent set method for deadlock detection.

### 5.1 Step Executions in Foata Normal Form for 1-Safe Nets

In [1] the authors showed that there exists a bijective function between the equivalence classes of occurrence sequences and the equivalence classes of processes for finite synchronization systems. A process exposes a partial ordering on the set of its conditions and events, which are labeled by places and transitions in the net, respectively. Two processes are equivalent iff they can be transformed to each other by the operations of ‘swap’ of the concurrent conditions that are labeled by the same place [1]. In other words, the differences of processes in the same equivalence class lie in that the individualities of several tokens on the same place are distinguished. The CTS semantics ignores the differences and makes an equivalence class of occurrence sequences compressed into one canonical TSS. Here note that the two kinds of equivalence classes of occurrence sequences (i.e., in the sense of [1] and the CTS semantics) may be quite different. An example is given in Fig. 5

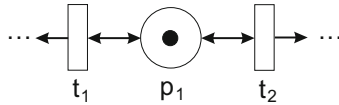


Fig. 5. A net system with initial marking  $M_0 = (1 \dots)$

Consider the occurrence sequence  $\sigma = t_1 t_2$  (markings omitted). By CTS semantics, it has an equivalent sequence  $\sigma' = t_2 t_1$  since both have the canonical TSS  $\bar{\delta} = \{t_1, t_2\}$ . However, they are not equivalent in the sense of [1] because  $t_1, t_2$  cannot be concurrently enabled by  $M_0$ . But when place  $p_1$  has two tokens (i.e.,  $M_0 = (2 \dots)$ ), the two

sequences are equivalent in both senses. Further, if the arrow from  $t_1$  to  $p_1$  is deleted (and  $\mathbf{M}_0 = (2 \cdots)$ ), then the result goes towards the opposite: the occurrence sequence  $t_1 t_2$  and  $t_2 t_1$  are not equivalent in CTS semantics but equivalent in the sense of [11].

Therefore, the result depends on two aspects: the number of tokens in places and self-loops in the net. Therefore, if the net under consideration is restricted to the class of pure 1-safe nets, the result may become very positive. In [11], the equivalence classes of occurrence sequences are based on a relation  $\equiv_0$  and two occurrence sequences  $\sigma_1 = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_i t_i \mathbf{M}_{i+1} t_{i+1} \cdots \mathbf{M}_k t_k$ ,  $\sigma_2 = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_i t_{i+1} \mathbf{M}'_{i+1} t_i \cdots \mathbf{M}_k t_k$  are in  $\equiv_0$  iff  $(\bullet t_i \cup t_i^\bullet) \cap (\bullet t_{i+1} \cup t_{i+1}^\bullet) = \emptyset$  for 1-safe nets. Then we have the theorem below:

**Theorem 7.** *Let  $\sigma_1 = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_i t_i \mathbf{M}_{i+1} t_{i+1} \cdots \mathbf{M}_k t_k$ ,  $\sigma_2 = \mathbf{M}_0 t_0 \mathbf{M}_1 t_1 \cdots \mathbf{M}_i t_{i+1} \mathbf{M}'_{i+1} t_i \cdots \mathbf{M}_k t_k$  be two occurrence sequences of a pure 1-safe net, then  $\sigma_1 \equiv_0 \sigma_2$  iff  $\sigma_1 \cong \sigma_2$ .*

*Proof.* By Definition 3, it is equivalent to prove:  $(\bullet t_i \cup t_i^\bullet) \cap (\bullet t_{i+1} \cup t_{i+1}^\bullet) = \emptyset$  iff  $(t_i, t_{i+1}) \in \text{order-ir}$ . For pure nets, the order-ir relation is just the strong order-ir, then by Property 1 it is sufficient to prove:  $\bullet t_i \cap \bullet t_{i+1} = \emptyset$  implies  $(\bullet t_i \cup t_i^\bullet) \cap (\bullet t_{i+1} \cup t_{i+1}^\bullet) = \emptyset$ . Note that both  $t_i, t_{i+1}$  are enabled by  $\mathbf{M}_i$ , then for  $\forall p \in \bullet t_i, \forall p' \in \bullet t_{i+1}$  it holds that  $p \neq p' \wedge \mathbf{M}_i(p) = \mathbf{M}_i(p') = 1$ . If there exists a place  $p$  such that  $p \in \bullet t_i \cap t_{i+1}^\bullet$  (i.e.,  $F(p, t_i) = F(t_{i+1}, p) = 1$ ), then  $\mathbf{M}'_{i+1}(p) = \mathbf{M}_i(p) - F(p, t_{i+1}) + F(t_{i+1}, p) = 2$  (because the net is pure,  $F(t_{i+1}, p) = 1$  implies  $F(p, t_{i+1}) = 0$ ), contradicting 1-safeness. Then  $\bullet t_i \cap t_{i+1}^\bullet = \emptyset$ . Similarly,  $\bullet t_{i+1} \cap t_i^\bullet = \emptyset$ .  $t_{i+1}^\bullet \cap t_i^\bullet = \emptyset$  holds for the same reason: if there exists a place  $p$  in  $t_{i+1}^\bullet \cap t_i^\bullet$  then it will have two more tokens after both  $t_i$  and  $t_{i+1}$  fire (i.e., at  $\mathbf{M}_{i+2}$ ). This completes the proof.  $\square$

Then for pure 1-safe nets, all the equivalence classes of occurrence sequences of finite length in [11] coincide with these based on the CTS semantics. In [11], each equivalence class corresponds to a process and in the CTS semantics all the occurrence sequences in the class are mapped to their common canonical TSS. Then there is a bijective function between the processes and canonical TSS's. Moreover, a construction for how to get the corresponding canonical TSS from a process can be found in [8].

When the net system is 1-safe, there is a construction in [8] to get step executions in Foata normal form from a process. A step for 1-safe nets at a marking  $\mathbf{M}$  corresponds to an execution of one transition set, in which all the transitions are concurrently enabled at  $\mathbf{M}$ . This coincides with the definition of EET if the net is a pure one. Correspondingly, a step execution in Foata normal form coincides with a canonical TSS. Intuitively, step semantics is an uncomplete combination of transitions. It avoids the computation of UC sets and meanwhile includes more branches in the exploring of state space. We will give some formal explanations in the following.

**Lemma 3.** *Every step execution is a TSS for pure 1-safe nets, and vice versa.*

Below is the version of definition of step executions in Foata normal form for 1-safe nets from [8].

**Definition 11.** *A step execution  $\delta = \mathbf{M}_0 T_0 \mathbf{M}_1 T_1 \cdots \mathbf{M}_n T_n$  is in Foata normal form iff:*

- $\delta = \varepsilon$  (i.e.,  $\delta$  is an empty one), or
- for each  $1 \leq i \leq n$  and for each  $t \in T_i$  there exists a transition  $t'$  in  $T_{i-1}$  such that  $t' \bullet \cap \bullet t \neq \emptyset$ .

We show in the following that for pure 1-safe nets the notion of step execution in Foata normal form coincides with that of canonical TSS, and then we can get an experimental support for the CTS semantics.

**Theorem 8.** *For pure 1-safe nets, a step execution  $\delta = M_0 T_0 M_1 T_1 \cdots M_n T_n$  is in Foata normal form iff it is a canonical TSS.*

*Proof.*  $\Rightarrow$ : The case for  $\delta = \varepsilon$  is trivial. Assume in Definition [11](#)  $p \in t' \bullet \cap \bullet t$ , then  $p \notin \bullet t'$ . If  $t$  is enabled at  $M_{i-1}$  then  $p$  has one token at  $M_{i-1}$ . If we choose  $t'$  to fire at  $M_{i-1}$  then  $p$  gets another token because  $p \notin \bullet t' \wedge p \in t' \bullet$ , which contradicts 1-safeness. Therefore,  $t$  is not enabled at  $M_{i-1}$  and condition [11](#) is not satisfied. Since this is for all the  $i$  with  $1 \leq i \leq n$ , then  $\delta$  is a canonical TSS.

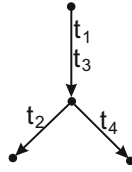
$\Leftarrow$ : It is trivially true for  $\delta = \varepsilon$ . If  $\delta$  is a canonical TSS, then for each  $i$  ( $1 \leq i \leq n$ ) and for each  $t \in T_i$ , condition [11](#) is not satisfied. If  $t$  is enabled at  $M_{i-1}$ , then there exists a transition  $t'$  in  $T_{i-1}$  such that  $(t, t') \notin \text{order-ir}$ . Since the net is pure, then there exists a place  $p$  such that  $p \in \bullet t \cap \bullet t'$ ; since  $t$  and  $t'$  are enabled at  $M_{i-1}$ , then  $p$  is marked at  $M_{i-1}$ . Because the transitions in  $T_{i-1}$  are order irrelevant, we can make  $t'$  fire at last. Then the occurrence of  $t'$  consumes the token in  $p$ , and it can not generate a new one to  $p$  because of the pureness of the net. This contradicts the enabledness of  $t$  at  $M_{i-1}$ . Therefore,  $t$  is not enabled at  $M_{i-1}$ , then some place  $p \in \bullet t$  is not marked at  $M_{i-1}$ . However,  $p$  is marked again at  $M_i$ , which indicates that there exists a transition  $t'$  in  $T_{i-1}$  whose occurrence generates a token in  $p$ , i.e.,  $p \in t' \bullet \cap \bullet t$ .  $\square$

Thanks to Theorem [8](#), we do not distinguish the notions *canonical TSS* and *step execution in Foata normal form* for pure 1-safe nets. In [\[8\]](#) Heljanko transformed the bounded reachability problem for 1-safe nets (most experimental cases are for pure nets) into constrained Boolean circuit satisfiability and made some experimental observations that CTS gives a better performance than step executions (not necessarily in Foata normal form) and interleaving semantics (i.e., only one transition fires for each step): CTS and step semantics find states in less steps than interleaving semantics; CTS finds states in less time than step semantics.

## 5.2 Deadlock Detection Based on Covering Step Graph and Persistent Sets

The notion of Covering step graph was introduced in [\[16\]](#). It can be combined with the persistent set method (called PSG) to detect deadlocks [\[14\]](#). A hybrid PSG shows more efficient performance than others according to the experimental analysis in [\[14\]](#). We do not give details on how this method works, and more can be referred to in [\[14\]](#). However, an example is given to illustrate the differences between it and the CTS method for deadlock detection. The hybrid PSG for the net in Fig. [1](#) is shown in Fig. [6](#). A comparison is drawn between Fig. [6](#) and Fig. [4](#), and we can find that:

1. The hybrid PSG always puts common conflict-free (conflict with no transition) transitions forwards to fire, such as the step  $\{t_1, t_3\}$  in Fig. [6](#).



**Fig. 6.** Hybrid PSG for the net in Fig. 1

2. The  $PS S(M)$  exploration of CTS semantics manages to reach all the deadlocks by using as few steps as possible, such as the deadlock by one step  $\{t_1, t_2, t_3\}$  in Fig. 4

The latter point is actually the most essential feature of the CTS semantics. The overhead of CTS semantics for deadlock detection is apparent: it cost much to compute the UC sets and the persistent sets. It seems a good suggestion that it can be applied in an uncomplete way, such as the bounded model checking, to achieve an effective use of its short steps to search for one deadlock.

## 6 Conclusions

We propose a semantics of canonical transition set sequence for Petri nets and give proofs for its well-definedness, soundness and completeness, then a bijection between the canonical transition set (CTS) sequences and occurrence sequences of finite length is found, by which a complete state space exploring method is proposed if the reachability set of the net system is finite. Furthermore, for the analysis of deadlocks, a method combining the CTS semantics and the persistent set method is proposed to efficiently detect deadlocks. For pure 1-safe nets, we find that CTS coincides with the step semantics in Foata normal form [8], which gives an experimental support for our method. We find the most notable feature of CTS semantics lies in that it makes state exploration to reach a (dead) state within a very few steps.

In the future, we may resort to bounded model checking method for the analysis of system behaviors based on CTS. Then deadlocks reached in very few steps can be detected quickly. We will also consider more properties to be checked and try to deal with infinite occurrence sequences.

**Acknowledgment.** The authors would like to thank the anonymous referees for their valuable comments and suggestions, which led to a substantial improvement of this paper.

## References

1. Best, E., Devillers, R.: Sequential and Concurrent Behaviour in Petri Net Theory. Theoretical Computer Science 55(1), 87–136 (1987)
2. Davillers, R., Janicki, R., Koutny, M., Lauer, P.E.: Concurrent and Maximally Concurrent Evolution of Non-sequential Systems. Theoretical Computer Science 43, 213–238 (1986)

3. Diekert, V., Metivier, Y.: Partial commutation and traces. In: Handbook of formal languages, vol. 3, pp. 457–534. Springer, Berlin (1997)
4. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
5. Girault, C., Valk, R.: Petri nets for systems engineering: A guide to modeling, verification, and applications. Springer, Heidelberg (2003)
6. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design* 2(2), 149–164 (1993)
7. Heljanko, K.: Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamental Informaticae* 37(3), 247–268 (1999)
8. Heljanko, K.: Bounded reachability checking with process semantics. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 218–232. Springer, Heidelberg (2001)
9. Heljanko, K., Niemela, I.: Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming* 3(4), 519–550 (2003)
10. Hoogeboom, H.J., Rozenberg, G.: Diamond properties of elementary net systems. *Fundamental Informaticae* 14(3), 287–300 (1991)
11. Mazurkiewicz, A.: Trace Theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1987)
12. Murata, T.: Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE* 77(4), 541–580 (1989)
13. Reisig, W.: Petri nets: An introduction. Springer, Heidelberg (1985)
14. Ribet, P.O., Vernadat, F., Berthomieu, B.: On combining the persistent sets method with the covering steps graph method. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 344–359. Springer, Heidelberg (2002)
15. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
16. Vernadat, F., Azema, P., Michel, F.: Covering step graph. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 516–535. Springer, Heidelberg (1996)
17. Wolper, P., Godefroid, P.: Partial-order methods for temporal verification. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 233–246. Springer, Heidelberg (1993)



# A Characterization of Combined Traces Using Labeled Stratified Order Structures

Dai Tri Man Lê

Department of Computer Science, University of Toronto  
10 King's College Road, Toronto, ON, M5S 3G4 Canada  
ledt@cs.toronto.edu

**Abstract.** This paper defines a class of labeled stratified order structures that characterizes exactly the notion of *combined traces* (i.e., *comtraces*) proposed by Janicki and Koutny in 1995. Our main technical contributions are the representation theorems showing that comtrace quotient monoid, *combined dependency graph* (Kleijn and Koutny 2008) and our labeled stratified order structure characterization are three different and yet equivalent ways to represent comtraces.

**Keywords:** Causality theory of concurrency, combined traces monoids, step sequences, stratified order structures, label-preserving isomorphism.

## 1 Introduction

Partial orders are a principle tool for modelling “true concurrency” semantics of concurrent systems (cf. [29]). They are utilized to develop powerful partial-order based automatic verification techniques, e.g., *partial order reduction* for model checking concurrent software (see, e.g., [1, Chapter 10] and [8]). Partial orders are also equipped with *traces*, their powerful formal language counterpart, proposed by Mazurkiewicz [28]. In *The Book of Traces* [5], trace theory has been used to tackle problems from diverse areas including formal language theory, combinatorics, graph theory, algebra, logic, and *concurrency theory*.

However, while partial orders and traces can sufficiently model the “earlier than” relationship, Janicki and Koutny argued that it is problematic to use a single partial order to specify both the “earlier than” and the “not later than” relationships [17]. This motivates them to develop the theory of *relational structures*, where a pair of relations is used to capture concurrent behaviors. The most well-known among the classes of relational structures proposed by Janicki and Koutny is the class of *stratified order structures* (*so-structures*) [12,16,18,19,15]. A so-structure is a triple  $(X, \prec, \sqsupseteq)$ , where  $\prec$  and  $\sqsupseteq$  are binary relations on  $X$ . They were invented to model both the “earlier than” (the relation  $\prec$ ) and “not later than” (the relation  $\sqsupseteq$ ) relationships, under the assumption that system runs are described by *stratified partial orders*, i.e., step sequences. They have been successfully applied to model inhibitor and priority systems, asynchronous races, synthesis problems, etc. (see for example [18,20,24,23,25,26] and others).

The *combined trace* (*comtrace*) notion, introduced by Janicki and Koutny [18], generalizes the trace notion by utilizing step sequences instead of words. First the set of

all possible steps that generates step sequences are identified by a relation *sim*, which is called *simultaneity*. Second a congruence relation is determined by a relation *ser*, which is called *serializability* and in general *not* symmetric. Then a comtrace is defined as a finite set of congruent step sequences. Comtraces were introduced as a formal language representation of so-structures to provide an operational semantics for Petri nets with inhibitor arcs. Unfortunately, comtraces have been less often known and applied than so-structures, even though in many cases they appear to be more natural. We believe one reason is that the comtrace notion was too succinctly discussed in [18] without a full treatment dedicated to comtrace theory. Motivated by this, Janicki and the author have devoted our recent effort on the study of comtraces [21,27,22], yet there are too many different aspects to explore and the truth is we can barely scratch the surface. In particular, the huge amount of results from trace theory (e.g., from [5,6]) desperately needs to be generalized to comtraces. These tasks are often non-trivial since we are required to develop intuition and novel techniques to deal with the complex interactions of the “earlier than” and “not later than” relations.

This paper gives a novel characterization of comtraces using labeled so-structures. Such definition is interesting for the following reasons.

First, it defines exactly the class of labeled so-structures that can be represented by comtraces. It is worth noting that this point is particularly important. Even though it was shown in [18] that every comtrace can be represented by a labeled so-structure, the converse could not be shown because a class of labeled so-structures that defines precisely the class of comtraces was not known. The closest to our characterization is the *combined dependency graph* (*cd-graph*) notion (analogous to *dependence graph* representation of traces) introduced recently by Kleijn and Koutny [26], but again a theorem showing that combined dependency graphs can be represented by comtraces was not given. Our approach is quite different and based on some new ideas discussed in Section 4 of this paper.

Second, even though the step sequence definition of comtraces is more suitable when dealing with formal language aspects of comtraces, the labeled so-structure representation is more suitable for a variety of powerful order-theoretic results and techniques available to us (cf. [11,3,15]).

Finally, the labeled so-structure definition of comtrace can be easily extended to *infinite comtraces*, which describe nonterminating concurrent processes. The labeled poset representation of infinite traces is already successfully applied in both theory and practice, e.g., [31,9,10,14]. Although such definition is equivalent to the one using quotient monoid over infinite words [13,4], we believe that infinite labeled posets are sometimes simpler. Indeed the celebrated work by Thiagarajan and Walukiewicz (cf. [31]) on linear temporal logic for traces utilizes the labeled poset characterization of infinite traces, where *configurations* of a trace are conveniently defined as *finite downward closed subsets* of the labeled poset representation. We will not analyze infinite comtraces or logics for comtraces in this paper, but these are fruitful directions to explore using the results from this paper.

The paper is organized as follows. In Section 2, we recall some preliminary definitions and notations. In Section 3, we give a concise exposition of the theory of so-structures and comtraces by Janicki and Koutny [18,19]. In Section 4, we give our

definition of comtraces using labeled so-structure and some remarks on how we arrived at such definition. In Section 5, we prove a representation theorem showing that our comtrace definition and the one by Janicki and Koutny are indeed equivalent; then using this theorem, we prove another representation theorem showing that our definition is also equivalent to the cd-graph definition from [26]. In Section 6, we define *composition* operators for our comtrace representation and for cd-graphs. Finally, in Section 7, some final remarks and future works are presented.

## 2 Notations

### 2.1 Relations, Orders and Equivalences

The *powerset* of a set  $X$  will be denoted by  $\wp(X)$ , i.e.  $\wp(X) \triangleq \{Y \mid Y \subseteq X\}$ . The set of all *non-empty* subsets of  $X$  will be denoted by  $\wp^{\setminus\{\emptyset\}}(X)$ . In other words,  $\wp^{\setminus\{\emptyset\}}(X) \triangleq \wp(X) \setminus \{\emptyset\}$ .

We let  $id_X$  denote the *identity relation* on a set  $X$ . If  $R$  and  $S$  are binary relations on a set  $X$  (i.e.,  $R, S \subseteq X \times X$ ), then their *composition*  $R \circ S$  is defined as  $R \circ S \triangleq \{(x, y) \in X \times X \mid \exists z \in X. (x, z) \in R \wedge (z, y) \in S\}$ . We also define

$$R^0 \triangleq id_X \quad R^i \triangleq R^{i-1} \circ R \quad (\text{for } i \geq 1) \quad R^+ \triangleq \bigcup_{i \geq 1} R^i \quad R^* \triangleq \bigcup_{i \geq 0} R^i$$

The relations  $R^+$  and  $R^*$  are called the (*irreflexive*) *transitive closure* and *reflexive transitive closure* of  $R$  respectively.

A binary relation  $R \subseteq X \times X$  is an *equivalence relation* on  $X$  if and only if (iff)  $R$  is *reflexive*, *symmetric* and *transitive*. If  $R$  is an equivalence relation, then for every  $x \in X$ , the set  $[x]_R \triangleq \{y \mid y R x \wedge y \in X\}$  is the equivalence class of  $x$  with respect to  $R$ . We also define  $X/R \triangleq \{[x]_R \mid x \in X\}$ , i.e., the set of all equivalence classes of  $X$  under  $R$ . We drop the subscript and write  $[x]$  when  $R$  is clear from the context.

A binary relation  $\prec \subseteq X \times X$  is a *partial order* iff  $R$  is *irreflexive* and *transitive*. The pair  $(X, \prec)$  in this case is called a *partially ordered set* (*poset*). The pair  $(X, \prec)$  is called a *finite poset* if  $X$  is finite. For convenience, we define:

$$\begin{aligned} \simeq_{\prec} &\triangleq \{(a, b) \in X \times X \mid a \not\prec b \wedge b \not\prec a\} && (\text{incomparable}) \\ \frown_{\prec} &\triangleq \{(a, b) \in X \times X \mid a \simeq_{\prec} b \wedge a \neq b\} && (\text{distinctly incomparable}) \\ \prec^{\frown} &\triangleq \{(a, b) \in X \times X \mid a \prec b \vee a \frown_{\prec} b\} && (\text{not greater}) \end{aligned}$$

A poset  $(X, \prec)$  is *total* iff  $\frown_{\prec}$  is empty; and *stratified* iff  $\simeq_{\prec}$  is an equivalence relation. Evidently every total order is stratified.

### 2.2 Step Sequences

For every finite set  $X$ , a set  $\mathbb{S} \subseteq \wp^{\setminus\{\emptyset\}}(X)$  can be seen as an alphabet. The elements of  $\mathbb{S}$  are called *steps* and the elements of  $\mathbb{S}^*$  are called *step sequences*. For example, if the set of possible steps is  $\mathbb{S} = \{\{a, b, c\}, \{a, b\}, \{a\}, \{c\}\}$ , then  $\{a, b\}\{c\}\{a, b, c\} \in \mathbb{S}^*$  is a

step sequence. The triple  $(\mathbb{S}^*, *, \varepsilon)$ , where  $\_ * \_$  denotes the step sequence concatenation operator (usually omitted) and  $\varepsilon$  denotes the empty step sequence, is a monoid.

Let  $t = A_1 \dots A_k$  be a step sequence. We define  $|t|_a$ , the number of occurrences of an event  $a$  in  $w$ , as  $|t|_a \triangleq |\{A_i \mid 1 \leq i \leq k \wedge a \in A_i\}|$ , where  $|X|$  denotes the cardinality of the set  $X$ . Then we can construct its unique *enumerated step sequence*  $\bar{t}$  as

$$\bar{t} \triangleq \overline{A_1} \dots \overline{A_k}, \text{ where } \overline{A_i} \triangleq \left\{ e^{(|A_1 \dots A_{i-1}|_e + 1)} \mid e \in A_i \right\}.$$

We will call such  $\alpha = e^{(j)} \in \overline{A_i}$  an *event occurrence* of  $e$ . For instance, if we let  $t = \{a, b\}\{b, c\}\{c, a\}\{a\}$ , then  $\bar{t} = \{a^{(1)}, b^{(1)}\}\{b^{(2)}, c^{(1)}\}\{a^{(2)}, c^{(2)}\}\{a^{(3)}\}$ .

We let  $\Sigma_t = \bigcup_{i=1}^k \overline{A_i}$  denote the set of all event occurrences in all steps of  $t$ . For example, when  $t = \{a, b\}\{b, c\}\{c, a\}\{a\}$ ,  $\Sigma_t = \{a^{(1)}, a^{(2)}, a^{(3)}, b^{(1)}, b^{(2)}, c^{(1)}, c^{(2)}\}$ . We also define  $\ell : \Sigma_t \rightarrow E$  to be the function that returns the *label* of  $\alpha$  for each  $\alpha \in \Sigma_t$ . For example, if  $\alpha = e^{(j)}$ , then  $\ell(\alpha) = \ell(e^{(j)}) = e$ . Hence, from an enumerated step sequence  $\bar{t} = \overline{A_1} \dots \overline{A_k}$ , we can uniquely reconstruct its step sequence  $t = \ell(\overline{A_1}) \dots \ell(\overline{A_k})$ .

For each  $\alpha \in \Sigma_u$ , we let  $pos_t(\alpha)$  denote the consecutive number of a step where  $\alpha$  belongs, i.e., if  $\alpha \in \overline{A_i}$  then  $pos_t(\alpha) = i$ . For our example,  $pos_t(a^{(2)}) = 3$ ,  $pos_t(b^{(2)}) = pos_t(c^{(1)}) = 2$ , etc.

It is important to observe that step sequences and stratified orders are interchangeable concepts. Given a step sequence  $u$ , define the binary relation  $\triangleleft_u$  on  $\Sigma_u$  as

$$\alpha \triangleleft_u \beta \stackrel{df}{\iff} pos_u(\alpha) < pos_u(\beta).$$

Intuitively,  $\alpha \triangleleft_u \beta$  simply means  $\alpha$  occurs before  $\beta$  on the step sequence  $u$ . Thus,  $\alpha \triangleleft_u \widehat{\beta}$  iff  $(\alpha \neq \beta \wedge pos_u(\alpha) \leq pos_u(\beta))$ ; and  $\alpha \simeq_u \beta$  iff  $pos_u(\alpha) = pos_u(\beta)$ . Obviously, the relation  $\triangleleft_u$  is a stratified order and we will call it the stratified order *generated by the step sequence*  $u$ .

Conversely, let  $\triangleleft$  be a stratified order on a set  $\Sigma$ . The set  $\Sigma$  can be represented as a sequence of equivalence classes  $\Omega_{\triangleleft} = B_1 \dots B_k$  ( $k \geq 0$ ) such that

$$\triangleleft = \bigcup_{i < j} B_i \times B_j \quad \text{and} \quad \simeq_{\triangleleft} = \bigcup_i B_i \times B_i.$$

The sequence  $\Omega_{\triangleleft}$  is called the step sequence *representing*  $\triangleleft$ . A detailed discussion on this connection between stratified orders and step sequences can be found in [22].

### 3 Stratified Order Structures and Combined Traces

In this section, we review the Janicki – Koutny theory of stratified order structures and comtraces from [18][19]. The reader is also referred to [26] for an excellent introductory survey on the subject with many motivating examples.

#### 3.1 Stratified Order Structures

A *relational structure* is a triple  $T = (X, R_1, R_2)$ , where  $X$  is a set and  $R_1, R_2$  are binary relations on  $X$ . A relational structure  $T' = (X', R'_1, R'_2)$  is an *extension* of  $T$ , denoted as  $T \subseteq T'$ , iff  $X = X'$ ,  $R_1 \subseteq R'_1$  and  $R_2 \subseteq R'_2$ .

**Definition 1 (stratified order structure [19]).** A stratified order structure (so-structure) is a relational structure  $S = (X, \prec, \sqsubseteq)$ , such that for all  $\alpha, \beta, \gamma \in X$ , the following hold:

$$\begin{array}{ll} \mathbf{S1:} & \alpha \not\prec \alpha \\ \mathbf{S2:} & \alpha \prec \beta \implies \alpha \sqsubseteq \beta \\ \mathbf{S3:} & \alpha \sqsubseteq \beta \sqsubseteq \gamma \wedge \alpha \neq \gamma \implies \alpha \prec \gamma \\ \mathbf{S4:} & \alpha \sqsubseteq \beta \prec \gamma \vee \alpha \prec \beta \sqsubseteq \gamma \implies \alpha \prec \gamma \end{array}$$

When  $X$  is finite,  $S$  is called a finite so-structure. ■

The axioms **S1–S4** imply that  $\prec$  is a partial order and  $\alpha \prec \beta \implies \beta \not\prec \alpha$ . The axioms **S1** and **S3** imply  $\sqsubseteq$  is a *strict preorder*. The relation  $\prec$  is called *causality* and represents the “earlier than” relationship while the relation  $\sqsubseteq$  is called *weak causality* and represents the “not later than” relationship. The axioms **S1–S4** model the mutual relationship between “earlier than” and “not later than” relations, provided that *the system runs are stratified orders*. Historically, the name “stratified order structure” came from the fact that stratified orders can be seen as a special kind of so-structures.

**Proposition 1 ([17]).** For every stratified poset  $(X, \triangleleft)$ , the triple  $S_{\triangleleft} = (X, \triangleleft, \triangleleft^{\circ})$  is a so-structure. □

We next recall the notion of *stratified order extension*. This concept is extremely important since the relationship between stratified orders and so-structures is exactly analogous to the one between total orders and partial orders.

**Definition 2 (stratified extension [19]).** Let  $S = (X, \prec, \sqsubseteq)$  be a so-structure. A stratified order  $\triangleleft$  on  $X$  is a stratified extension of  $S$  if and only if  $(X, \prec, \sqsubseteq) \subseteq (X, \triangleleft, \triangleleft^{\circ})$ .

The set of all stratified extensions of  $S$  is denoted as  $\text{ext}(S)$ . ■

Szpilrajn’s Theorem [30] states that every poset can be reconstructed by taking the intersection of all of its total order extensions. Janicki and Koutny showed that a similar result holds for so-structures and stratified extensions:

**Theorem 1 ([19]).** Let  $S = (X, \prec, \sqsubseteq)$  be a so-structure. Then

$$S = \left( X, \bigcap_{\triangleleft \in \text{ext}(S)} \triangleleft, \bigcap_{\triangleleft \in \text{ext}(S)} \triangleleft^{\circ} \right). \quad \square$$

Using this theorem, we can show the following properties relating so-structures and their stratified extensions.

**Corollary 1.** For every so-structure  $S = (X, \prec, \sqsubseteq)$ ,

1.  $(\exists \triangleleft \in \text{ext}(S), \alpha \triangleleft \beta) \wedge (\exists \triangleleft \in \text{ext}(S), \beta \triangleleft \alpha) \implies (\exists \triangleleft \in \text{ext}(S), \beta \triangleleft^{\circ} \alpha)$ .
2.  $(\forall \triangleleft \in \text{ext}(S), \alpha \triangleleft \beta \vee \beta \triangleleft \alpha) \iff \alpha \prec \beta \vee \beta \prec \alpha$ .

*Proof.* **1.** See [19] Theorem 3.6]. **2.** Follows from **1.** and Theorem □ □

### 3.2 Combined Traces

*Comtraces* were introduced in [18] as a generalization of traces to represent so-structures. The *comtrace congruence* is defined via two relations *simultaneity* and *serializability*.

**Definition 3 (comtrace alphabet [18]).** Let  $E$  be a finite set (of events) and let  $ser \subseteq sim \subseteq E \times E$  be two relations called serializability and simultaneity respectively and the relation  $sim$  is irreflexive and symmetric. The triple  $\theta = (E, sim, ser)$  is called a comtrace alphabet. ■

Intuitively, if  $(a, b) \in sim$  then  $a$  and  $b$  can occur simultaneously (or be a part of a *synchronous* occurrence in the sense of [24]), while  $(a, b) \in ser$  means that  $a$  and  $b$  may occur simultaneously or  $a$  may occur before  $b$ . We define  $\mathbb{S}_\theta$ , the set of all possible *steps*, to be the set of all cliques of the graph  $(E, sim)$ , i.e.,

$$\mathbb{S}_\theta \triangleq \{A \mid A \neq \emptyset \wedge \forall a, b \in A, (a = b \vee (a, b) \in sim)\}.$$

**Definition 4 (comtrace congruence [18]).** For a comtrace alphabet  $\theta = (E, sim, ser)$ , we define  $\approx_\theta \subseteq \mathbb{S}_\theta^* \times \mathbb{S}_\theta^*$  to be the relation comprising all pairs  $(t, u)$  of step sequences such that

$$t = wAz \quad \text{and} \quad u = wBCz,$$

where  $w, z \in \mathbb{S}_\theta^*$  and  $A, B, C$  are steps satisfying  $B \cup C = A$  and  $B \times C \subseteq ser$ .

We define comtrace congruence  $\equiv_\theta \triangleq (\approx_\theta \cup \approx_\theta^{-1})^*$ . We define the comtrace concatenation operator  $_ \otimes _$  as  $[r] \otimes [t] \triangleq [r * t]$ . The quotient monoid  $(\mathbb{S}^* / \equiv_\theta, \otimes, [\epsilon])$  is called the monoid of comtraces over  $\theta$ . ■

Note that since  $ser$  is irreflexive,  $B \times C \subseteq ser$  implies that  $B \cap C = \emptyset$ . We will omit the subscript  $\theta$  from the comtrace congruence  $\approx_\theta$ , and write  $\equiv$  and  $\approx$  when it causes no ambiguity. To shorten our notations, we often write  $[s]_\theta$  or  $[s]$  instead of  $[s]_{\equiv_\theta}$  to denote the comtrace generated by the step sequence  $s$  over  $\theta$ .

*Example 1.* Let  $E = \{a, b, c\}$  where  $a, b$  and  $c$  are three atomic operations, where

$$a : y \leftarrow x + y \qquad b : x \leftarrow y + 2 \qquad c : y \leftarrow y + 1$$

Assume simultaneous reading is allowed. Then only  $b$  and  $c$  can be performed simultaneously, and the simultaneous execution of  $b$  and  $c$  gives the same outcome as executing  $b$  followed by  $c$ . We can then define the comtrace alphabet  $\theta = (E, sim, ser)$ , where  $sim = \{\{b, c\}\}$  and  $ser = \{(b, c)\}$ . This yields  $\mathbb{S}_\theta = \{\{a\}, \{b\}, \{c\}, \{b, c\}\}$ . Thus,  $\mathbf{t} = [\{a\}\{b, c\}] = \{\{a\}\{b, c\}, \{a\}\{b\}\{c\}\}$  is a comtrace. But  $\{a\}\{c\}\{b\} \notin \mathbf{t}$ . ■

Even though traces are quotient monoids over sequences and comtraces are quotient monoids over step sequences, traces can be regarded as special kinds of comtraces when the relation  $ser = sim$ . For a more detailed discussion on this connection between traces and comtraces, the reader is referred to [22].

**Definition 5 ([18]).** Let  $u \in \mathbb{S}_\theta^*$ . We define the relations  $\prec_u, \sqsubset_u \subseteq \Sigma_u \times \Sigma_u$  as:

1.  $\alpha \prec_u \beta \iff \alpha \triangleleft_u \beta \wedge (\ell(\alpha), \ell(\beta)) \notin ser,$
2.  $\alpha \sqsubset_u \beta \iff \alpha \triangleleft_u \widehat{\beta} \wedge (\ell(\beta), \ell(\alpha)) \notin ser.$  ■

It is worth noting that the structure  $(\Sigma_u, \prec_u, \sqsubset_u, \ell)$  is exactly the *cd-graph* (cf. Definition 11) that represents the comtrace  $[u]$ . This gives us some intuition on how Koutny and Kleijn constructed the *cd-graph* definition in [26]. We also observe that  $(\Sigma_u, \prec_u, \sqsubset_u)$  is usually *not* a so-structure since  $\prec_u$  and  $\sqsubset_u$  describe only basic “local” causality and weak causality invariants of the event occurrences of  $u$  by considering pairwise serializable relationships of event occurrences. Hence,  $\prec_u$  and  $\sqsubset_u$  might not capture “global” invariants that can be inferred from S2–S4 of Definition 11. To ensure all invariants are included, we need the following  $\diamond$ -closure operator.

**Definition 6** ([18]). *For every relational structure  $S = (X, R_1, R_2)$  we define  $S^\diamond$  as*

$$S^\diamond \triangleq (X, (R_1 \cup R_2)^* \circ R_1 \circ (R_1 \cup R_2)^*, (R_1 \cup R_2)^* \setminus id_X). \quad \blacksquare$$

Intuitively  $\diamond$ -closure is a generalization of transitive closure for relations to relational structures. The motivation is that for appropriate relations  $R_1$  and  $R_2$  (see assertion (3) of Proposition 2), the relational structure  $(X, R_1, R_2)^\diamond$  is a so-structure. The  $\diamond$ -closure operator satisfies the following properties:

**Proposition 2** ([18]). *Let  $S = (X, R_1, R_2)$  be a relational structure.*

1. *If  $R_2$  is irreflexive then  $S \subseteq S^\diamond$ .*
2.  *$(S^\diamond)^\diamond = S^\diamond$ .*
3.  *$S^\diamond$  is a so-structure if and only if  $(R_1 \cup R_2)^* \circ R_1 \circ (R_1 \cup R_2)^*$  is irreflexive.*
4. *If  $S$  is a so-structure then  $S = S^\diamond$ .*
5. *If  $S$  be a so-structure and  $S_0 \subseteq S$ , then  $S_0^\diamond \subseteq S$  and  $S_0^\diamond$  is a so-structure. □*

**Definition 7.** *Given a step sequence  $u \in \mathbb{S}_\theta^*$  and its respective comtrace  $\mathbf{t} = [u] \in \mathbb{S}_\theta^* / \equiv$ , we define the relational structures  $S_{\mathbf{t}}$  as:*

$$S_{\mathbf{t}} = (\Sigma_{\mathbf{t}}, \prec_{\mathbf{t}}, \sqsubset_{\mathbf{t}}) \triangleq (\Sigma_u, \prec_u, \sqsubset_u)^\diamond. \quad \blacksquare$$

The relational structure  $S_{\mathbf{t}}$  is called the *so-structure defined by the comtrace  $\mathbf{t} = [u]$* , where  $\Sigma_{\mathbf{t}}$ ,  $\prec_{\mathbf{t}}$  and  $\sqsubset_{\mathbf{t}}$  are used to denote the event occurrence set, causality relation and weak causality relation induced by the comtrace  $\mathbf{t}$  respectively. The following nontrivial theorem and its corollary justifies the name by showing that step sequences in a comtrace  $\mathbf{t}$  are exactly stratified extension of the so-structure  $S_{\mathbf{t}}$ , and that  $S_{\mathbf{t}}$  is uniquely defined for the comtrace  $\mathbf{t}$  regardless of the choice of  $u \in \mathbf{t}$ .

**Theorem 2** ([18]). *For each  $\mathbf{t} \in E^* / \equiv_\theta$ , the relational structure  $S_{\mathbf{t}}$  is a so-structure and  $\text{ext}(S_{\mathbf{t}}) = \{\prec_u \mid u \in \mathbf{t}\}$ . □*

**Corollary 2.** *For all  $\mathbf{t}, \mathbf{q} \in E^* / \equiv_\theta$ ,*

1.  *$\mathbf{t} = \mathbf{q} \implies S_{\mathbf{t}} = S_{\mathbf{q}}$*
2.  *$S_{\mathbf{t}} = (\Sigma_{\mathbf{t}}, \prec_{\mathbf{t}}, \sqsubset_{\mathbf{t}}) = (\Sigma_{\mathbf{t}}, \bigcap_{w \in \mathbf{t}} \prec_w, \bigcap_{w \in \mathbf{t}} \prec_w^\wedge)$  □*

## 4 Comtraces as Labeled Stratified Order Structures

Even though Theorem 2 shows that each comtrace can be represented uniquely by a labeled so-structure, it does not give us an explicit definition of how these labeled so-structures look like. In this section, we will give an exact definition of labeled so-structures that represent comtraces. To provide us with more intuition, we first recall how Mazurkiewicz traces can be characterized as labeled posets.

A *trace concurrent alphabet* is a pair  $(E, ind)$ , where  $ind$  is a symmetric irreflexive binary relation on the finite set  $E$ . A *trace congruence*  $\equiv_{ind}$  can then be defined as the smallest equivalence relation such that for all sequences  $uabv, ubav \in E^*$ , if  $(a, b) \in ind$ , then  $uabv \equiv_{ind} ubav$ . The elements of  $E^* / \equiv_{ind}$  are called *traces*.

Traces can also be defined alternatively as posets whose elements are labeled with symbols of a concurrent alphabet  $(E, ind)$  satisfying certain conditions.

Given a binary relation  $R \subseteq X$ , the *covering relation* of  $R$  is defined as  $R^{cov} \triangleq \{(x, y) \mid x R y \wedge \neg \exists z, x R z R y\}$ . An alternative definition of Mazurkiewicz trace is:

**Definition 8 (cf. [31]).** A *trace over a concurrent alphabet*  $(E, ind)$  is a finite labeled poset  $(X, \prec, \lambda)$ , where  $\lambda : X \rightarrow E$  is a labeling function, such that for all  $\alpha, \beta \in X$ ,

1.  $\alpha \prec^{cov} \beta \implies (\lambda(\alpha), \lambda(\beta)) \notin ind$ , and
2.  $(\lambda(\alpha), \lambda(\beta)) \notin ind \implies \alpha \prec \beta \vee \beta \prec \alpha$ . ■

A trace in this definition is only identified unique up to *label-preserving isomorphism*. The first condition says that immediately causally related event occurrences must be labeled with dependent events. The second condition ensures that any two event occurrences with dependent labels must be causally related. The first condition is particularly important since two immediately causally related event occurrences will occur next to each other in at least one of its linear extensions. This is the key to relate Definition 8 with quotient monoid definition of traces. Thus, we would like to establish a similar relationship for comtraces. An immediate technical difficulty is that weak causality might be cyclic, so the notion of “immediate weak causality” does not make sense. However, we can still deal with cycles of a so-structure by taking advantage of the following simple fact: *the weak causality relation is a strict preorder*.

Let  $S = (X, \prec, \sqsubset)$  be a so-structure. We define the relation  $\equiv_{\sqsubset} \subseteq X \times X$  as

$$\alpha \equiv_{\sqsubset} \beta \stackrel{df}{\iff} \alpha = \beta \vee (\alpha \sqsubset \beta \wedge \beta \sqsubset \alpha)$$

Since  $\sqsubset$  is a strict preorder, it follows that  $\equiv_{\sqsubset}$  is an equivalence relation. The relation  $\equiv_{\sqsubset}$  will be called the  $\sqsubset$ -*cycle equivalence relation* and an element of the quotient set  $X / \equiv_{\sqsubset}$  will be called a  $\sqsubset$ -*cycle equivalence class*. We then define the following binary relations  $\hat{\succ}$  and  $\hat{\sqsubset}$  on the quotient set  $X / \equiv_{\sqsubset}$  as

$$[\alpha] \hat{\succ} [\beta] \stackrel{df}{\iff} ([\alpha] \times [\beta]) \cap \prec \neq \emptyset \quad \text{and} \quad [\alpha] \hat{\sqsubset} [\beta] \stackrel{df}{\iff} ([\alpha] \times [\beta]) \cap \sqsubset \neq \emptyset \quad (4.1)$$

Using this quotient construction, every so-structure, whose weak causality relation might be cyclic, can be uniquely represented by an *acyclic* quotient so-structure.



**Proposition 3.** *The relational structure  $S/\equiv_{\sqsubset} \triangleq (X/\equiv_{\sqsubset}, \hat{\prec}, \hat{\sqsubset})$  is a so-structure, the relation  $\hat{\sqsubset}$  is a partial order, and for all  $x, y \in X$ ,*

1.  $\alpha \prec \beta \iff [\alpha] \hat{\succ} [\beta]$
2.  $\alpha \sqsubset \beta \iff [\alpha] \hat{\sqsubset} [\beta] \vee (\alpha \neq \beta \wedge [\alpha] = [\beta])$

*Proof.* Follows from Definition [1](#). □

Using [\(4.1\)](#) and Theorem [1](#) it is not hard to prove the following simple yet useful properties of  $\sqsubset$ -cycle equivalence classes.

**Proposition 4.** *Let  $S = (X, \prec, \sqsubset)$  be a so-structure. We use  $u$  and  $v$  to denote some step sequences over  $\wp^{\setminus\{\emptyset\}}(X)$ . Then for all  $\alpha, \beta \in X$ ,*

1.  $[\alpha] = [\beta] \iff \forall \triangleleft \in \text{ext}(S), \alpha \simeq_{\triangleleft} \beta$
2.  $\exists \triangleleft \in \text{ext}(S), \Omega_{\triangleleft} = u[\alpha]v$
3.  $[\alpha] \hat{\sqsubset}^{\text{cov}} [\beta] \implies \exists \triangleleft \in \text{ext}(S), \Omega_{\triangleleft} = u[\alpha][\beta]v$  □

Each  $\sqsubset$ -cycle equivalence class is what Juhás, Lorenz and Mauser called a *synchronous step* [\[24,23\]](#). They also used equivalence classes to capture synchronous steps but only for the special class of *synchronous closed* so-structures, where  $(\sqsubset \setminus \prec) \cup \text{id}_X$  is an equivalence relation. We extend their ideas by using  $\sqsubset$ -cycle equivalence classes to capture what we will call *non-serializable sets* in arbitrary so-structures. The name is justified in assertion (1) of Proposition [4](#) stating that two elements belong to the same non-serializable set of a so-structure  $S$  iff they must be executed simultaneously in every stratified extension of  $S$ . Furthermore, we show in assertion (2) that all elements of a non-serializable set must occur together as a single step in at least one stratified extension of  $S$ . Assertion (3) gives a sufficient condition for two non-serializable sets to occur as consecutive steps in at least one stratified extension of  $S$ .

Before we proceed to define comtrace using labeled so-structure, we need to define *label-preserving isomorphisms* for labeled so-structures more formally. A tuple  $T = (X, P, Q, \lambda)$  is a *labeled relational structure* iff  $(X, P, Q)$  is a relational structure and  $\lambda$  is a function with domain  $X$ . If  $(X, P, Q)$  is a so-structure, then  $T$  is a *labeled so-structure*.

**Definition 9 (label-preserving isomorphism).** *Given two labeled relational structures  $T_1 = (X_1, P_1, Q_1, \lambda_1)$  and  $T_2 = (X_2, P_2, Q_2, \lambda_2)$ , we write  $T_1 \cong T_2$  to denote that  $T_1$  and  $T_2$  are label-preserving isomorphic (lp-isomorphic). In other words, there is a bijection  $f : X_1 \rightarrow X_2$  such that for all  $\alpha, \beta \in X_1$ ,*

1.  $(\alpha, \beta) \in P_1 \iff (f(\alpha), f(\beta)) \in P_2$
2.  $(\alpha, \beta) \in Q_1 \iff (f(\alpha), f(\beta)) \in Q_2$
3.  $\lambda_1(\alpha) = \lambda_2(f(\alpha))$

*Such function  $f$  is called a label-preserving isomorphism (lp-isomorphism).* ■

Note that all notations, definitions and results for so-structures are applicable to labeled so-structures. We also write  $[T]$  or  $[X, P, Q, \lambda]$  to denote the lp-isomorphic class of a labeled relational structure  $T = (X, P, Q, \lambda)$ . We will not distinguish an lp-isomorphic class  $[T]$  with a single labeled relational structure  $T$  when it does not cause ambiguity.

We are now ready to give an alternative definition for comtraces. To avoid confusion with the comtrace notion by Janicki and Koutny in [\[18\]](#), we will use the term *Isos-comtrace* to denote a comtrace defined using our definition.

**Definition 10 (Isos-comtrace).** Given a comtrace alphabet  $\theta = (E, sim, ser)$ , a Isos-comtrace over  $\theta$  is (an lp-isomorphic class of) a finite labeled so-structure  $[X, \prec, \sqsubset, \lambda]$  such that  $\lambda : X \rightarrow E$  and for all  $\alpha, \beta \in X$ ,

- LC1:**  $[\alpha](\hat{\sqsubset}^{cov} \cap \hat{\succ})[\beta] \implies \lambda([\alpha]) \times \lambda([\beta]) \notin ser$   
**LC2:**  $[\alpha](\hat{\sqsubset}^{cov} \setminus \hat{\succ})[\beta] \implies \lambda([\beta]) \times \lambda([\alpha]) \notin ser$   
**LC3:**  $\forall A, B \in \wp^{\neq \emptyset}([\alpha]), A \cup B = [\alpha] \implies \lambda(A) \times \lambda(B) \notin ser$   
**LC4:**  $(\lambda(\alpha), \lambda(\beta)) \notin ser \implies \alpha \prec \beta \vee \beta \sqsubset \alpha$   
**LC5:**  $(\lambda(\alpha), \lambda(\beta)) \notin sim \implies \alpha \prec \beta \vee \beta \prec \alpha$

We write  $LCT(\theta)$  to denote the class of all Isos-comtraces over  $\theta$ . ■

*Example 2.* Let  $E = \{a, b, c\}$ ,  $sim = \{\{a, b\}, \{a, c\}, \{b, c\}\}$  and  $ser = \{(a, b), (b, a), (a, c)\}$ . Then we have  $\mathbb{S} = \{\{a\}, \{b\}, \{c\}, \{b, c\}\}$ . The lp-isomorphic class of the labeled so-structure  $T = (X, \prec, \sqsubset, \lambda)$  depicted in Figure 1 (the dotted edges denote  $\sqsubset$  relation and the solid edges denote both  $\prec$  and  $\sqsubset$  relations) is a Isos-comtrace. The graph in Figure 2 represents the labeled quotient so-structure  $T/\equiv_{\sqsubset} = (X/\equiv_{\sqsubset}, \hat{\succ}, \hat{\sqsubset}, \lambda')$  of  $T$ , where we define  $\lambda'(A) = \{\lambda(x) \mid x \in A\}$ .

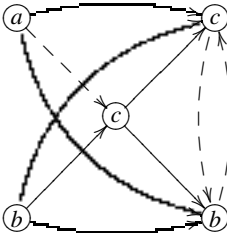


Fig. 1. Isos-comtrace  $[T]$

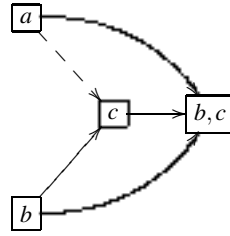


Fig. 2. the quotient structure  $T/\equiv_{\sqsubset}$  of  $T$

The Isos-comtrace  $[T]$  actually corresponds to the comtrace  $[\{a, b\}\{c\}\{b, c\}]$ , and we will show this relationship formally in Section 5. ■

*Remark 1.* Definition 10 can be extended to define *infinite comtrace* as follows. Instead of asking  $X$  to be finite, we require a labeled so-structure to be *initially finite* (cf. [19]), i.e.,  $\{\alpha \in X \mid \alpha \sqsubset \beta\}$  is finite for all  $\beta \in X$ . The initially-finiteness not only gives us a sensible interpretation that every event only causally depends on finitely many events, but also guarantees that the covering relations of  $\hat{\succ}$  and  $\hat{\sqsubset}$  are well-defined. ■

Since each Isos-comtrace is defined as a class of lp-isomorphic labeled so-structures, dealing with Isos-comtrace might seem tricky. Fortunately, the *auto-concurrency* property, i.e., the relation  $ser$  is irreflexive, gives us a *canonical* way to enumerate the events of a Isos-comtrace very similar to how the events of a comtrace are enumerated.

Given a step sequence  $s = A_1 \dots A_k$  and any function  $f$  defined on  $\bigcup_{i=1}^k A_i$ , we define  $\text{map}(f, s) \triangleq f(A_1) \dots f(A_k)$ , i.e., the step sequence derived from  $s$  by applying the function  $f$  successively on each  $A_i$ . Note that  $f(A_i)$  denotes the *image* of  $A_i$  under  $f$ .

Given a Isos-comtrace  $T = [X, \prec, \sqsubset, \lambda]$  over a comtrace alphabet  $\theta = (E, sim, ser)$ , a stratified order  $\triangleleft \in \text{ext}(T)$  can be seen as a step sequence  $\Omega_{\triangleleft} = A_1 \dots A_k$ .

**Proposition 5.** 1. For every  $i$  ( $1 \leq i \leq k$ ),  $|A_i| = |\lambda(A_i)|$

2.  $\text{map}(\lambda, \Omega_{\triangleleft}) = \lambda(A_1) \dots \lambda(A_k) \in \mathbb{S}_\theta^*$ . □

Proposition 5 ensures that  $u = \text{map}(\lambda, \Omega_{\triangleleft})$  is a valid step sequence over  $\theta$ . Recall that  $\bar{u} = \overline{A_1} \dots \overline{A_k}$  denotes the enumerated step sequence of  $u$  and  $\Sigma_u$  denotes the set of event occurrences. Define a bijection  $\xi_u : \Sigma_u \rightarrow X$  as

$$\xi_u(\alpha) = x \stackrel{\text{df}}{\iff} \alpha \in \overline{A_i} \wedge x \in A_i \wedge \lambda(x) = \ell(\alpha)$$

By Proposition 5 the function  $\xi_u$  is well-defined. Moreover, we can show that  $\xi_u$  is uniquely determined by  $T$  regardless of the choice of  $\triangleleft \in \text{ext}(T)$ .

**Proposition 6.** Given  $\triangleleft_1, \triangleleft_2 \in \text{ext}(T)$ , let  $v = \text{map}(\lambda, \Omega_{\triangleleft_1})$  and  $w = \text{map}(\lambda, \Omega_{\triangleleft_2})$ . Then  $\xi_v = \xi_w$ . □

Henceforth, we will ignore subscripts and reserve the notation  $\xi$  to denote the kind of mappings as defined above. We then define the *enumerated so-structure* of  $T$  to be the labeled so-structure  $T_0 = (\Sigma, \prec_0, \sqsubset_0, \ell)$ , where  $\Sigma = \Sigma_u$  for  $u = \text{map}(\lambda, \Omega_{\triangleleft})$  and  $\triangleleft \in \text{ext}(T)$ ; and the relations  $\prec_0, \sqsubset_0 \subseteq \Sigma \times \Sigma$  are defined as

$$\alpha \prec_0 \beta \stackrel{\text{df}}{\iff} \xi(\alpha) \prec \xi(\beta) \quad \text{and} \quad \alpha \sqsubset_0 \beta \stackrel{\text{df}}{\iff} \xi(\alpha) \sqsubset \xi(\beta)$$

Clearly, the enumerated so-structure  $T_0$  can be uniquely determined from  $T$  using the preceding definition. From our construction, we can easily show the following important relationships:

**Proposition 7.** 1.  $T_0$  and  $T$  are lp-isomorphic under the mapping  $\xi$ .

2. The labeled so-structures  $(\Sigma, \triangleleft_u, \triangleleft_u^\wedge, \ell)$  and  $(X, \triangleleft, \triangleleft^\wedge, \lambda)$  are lp-isomorphic under the mapping  $\xi$  and  $\triangleleft_u \in \text{ext}(T_0)$ . □

In other words, the mapping  $\xi : \Sigma \rightarrow X$  plays the role of both the lp-isomorphism from  $T_0$  to  $T$  and the lp-isomorphism from the stratified extension  $(\Sigma, \triangleleft_u)$  of  $T_0$  to the stratified extension  $(X, \triangleleft)$  of  $T$ . These relationships can be best captured using the commutative diagram on the right.

$$\begin{array}{ccc} (\Sigma, \prec_0, \sqsubset_0, \ell) & \xrightarrow{\xi} & (X, \prec, \sqsubset, \lambda) \\ \text{id}_\Sigma \downarrow & & \downarrow \text{id}_X \\ (\Sigma, \triangleleft_u, \triangleleft_u^\wedge, \ell) & \xrightarrow{\xi} & (X, \triangleleft, \triangleleft^\wedge, \lambda) \end{array}$$

We can even observe further that two Isos-comtraces are identical if and only if they define the same enumerated so-structure. Henceforth, we will call an enumerated so-structure defined by a Isos-comtrace  $T$  the *canonical representation* of  $T$ .

Recently, inspired by the dependency graph notion for Mazurkiewicz traces (cf. [5, Chapter 2]), Kleijn and Koutny claimed without proof that their *combined dependency graph* notion is another alternative way to define comtraces [26]. In Section 5 we will give a detailed proof of their claim.

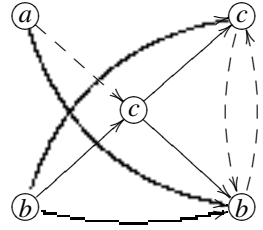
**Definition 11 (combined dependency graph [26]).** Given an comtrace alphabet  $\theta = (E, \text{ser}, \text{sim})$ , a combined dependency graph (cd-graph) over  $\theta$  is (a lp-isomorphic class of) a finite labeled relational structure  $D = [X, \longrightarrow, \dashrightarrow, \lambda]$  such that  $\lambda : X \rightarrow E$ , the relations  $\longrightarrow, \dashrightarrow$  are irreflexive,  $D^\diamond$  is a so-structure, and for all  $\alpha, \beta \in X$ ,

- CD1:  $(\lambda(\alpha), \lambda(\beta)) \notin \text{sim} \implies \alpha \longrightarrow \beta \vee \beta \longrightarrow \alpha$
- CD2:  $(\lambda(\alpha), \lambda(\beta)) \notin \text{ser} \implies \alpha \longrightarrow \beta \vee \beta \dashrightarrow \alpha$
- CD3:  $\alpha \longrightarrow \beta \implies (\lambda(\alpha), \lambda(\beta)) \notin \text{ser}$
- CD4:  $\alpha \dashrightarrow \beta \implies (\lambda(\beta), \lambda(\alpha)) \notin \text{ser}$

We will write  $\text{CDG}(\theta)$  to denote the class of all cd-graphs over  $\theta$ . ■

Cd-graphs can be seen as reduced graph-theoretic representations for Isos-comtraces, where some arcs that can be recovered using  $\diamond$ -closure are omitted. It is interesting to observe that the non-serializable sets of a cd-graph are exactly the *strongly connected components* of the directed graph  $(X, \dashrightarrow)$  and can easily be found in time  $O(|X| + |\dashrightarrow|)$  using any standard algorithm (cf. [2] Section 22.5).

*Remark 2.* Cd-graphs were called *dependence comdags* in [26]. But this name could be misleading since the directed graph  $(X, \dashrightarrow)$  is not necessarily acyclic. For example, the graph on the right is the cd-graph that corresponds to the Isos-comtrace from Figure 1, but it is not acyclic. (Here, we use the dotted edges to denote  $\dashrightarrow$  and the solid edges to denote *only*  $\longrightarrow$ .) Thus, we use the name “combined dependency graph” instead. ■



## 5 Representation Theorems

This section contains the main technical contribution of this paper by showing that for a given comtrace alphabet  $\theta$ ,  $\mathbb{S}^*/\equiv_\theta$ ,  $\text{LCT}(\theta)$  and  $\text{CDG}(\theta)$  are three equivalent ways of talking about the same class of objects. We will next prove the first representation theorem which establishes the representation mappings between  $\mathbb{S}^*/\equiv_\theta$  and  $\text{LCT}(\theta)$ .

### 5.1 Representation Theorem for Comtraces and Isos-Comtraces

**Proposition 8.** *Let  $S_0 = (X, \prec_0, \sqsubset_0)$  and  $S_1 = (X, \prec_1, \sqsubset_1)$  be stratified order structures such that  $\text{ext}(S_0) \subseteq \text{ext}(S_1)$ . Then  $S_1 \subseteq S_0$ .*

*Proof.* Follows from Theorem 1. □

For the next two lemmata, we let  $T$  be a Isos-comtrace over a comtrace alphabet  $\theta = (E, \text{sim}, \text{ser})$ . Let  $T_0 = (\Sigma_u, \prec_0, \sqsubset_0, \ell)$  be the canonical representation of  $T$ . Let  $\prec_0 \in \text{ext}(T_0)$  and  $u = \text{map}(\ell, \Omega_{\prec_0})$ . Since  $u$  is a valid step sequence in  $\mathbb{S}^*$  (by Proposition 5), we can construct  $S_{[u]} = (\Sigma_u, \prec_{[u]}, \sqsubset_{[u]})$  from Definition 7. Our goal is to show that the stratified order  $S_{[u]}$  defined by the comtrace  $[u]$  is exactly  $(\Sigma_u, \prec_0, \sqsubset_0)$ .

**Lemma 1.**  $S_{[u]} \subseteq (\Sigma_u, \prec_0, \sqsubset_0)$ .

*Proof.* By Proposition 2, to show  $S_{[u]} = (\Sigma_u, \prec_u, \sqsubset_u)^\diamond \subseteq (\Sigma_u, \prec_0, \sqsubset_0)$ , it suffices to show that  $(\Sigma_u, \prec_u, \sqsubset_u) \subseteq (\Sigma_u, \prec_0, \sqsubset_0)$ . Since  $T_0$  is the canonical representation of  $T$ , it is important to observe that  $\prec_0 = \prec_u$ .

( $\prec_u \subseteq \prec_0$ ): Assume  $\alpha \prec_u \beta$ . Then from Definition 5,  $\alpha \triangleleft_u \beta \wedge (\ell(\alpha), \ell(\beta)) \notin ser$ . Since  $(\ell(\alpha), \ell(\beta)) \notin ser$ , it follows from Definition 10 that  $\alpha \prec_0 \beta$  or  $\beta \sqsubset_0 \alpha$ . Suppose for a contradiction that  $\beta \sqsubset_0 \alpha$ , then by Theorem 1,  $\forall \triangleleft \in ext(T_0)$ ,  $\beta \triangleleft \alpha$ . But since we assume that  $\triangleleft_0 \in ext(T_0)$ , it follows that  $\triangleleft_u \in ext(T_0)$  and  $\alpha \triangleleft_u \beta$ , a contradiction. Hence, we have shown  $\alpha \prec_0 \beta$ .

( $\sqsubset_u \subseteq \sqsubset_0$ ): Can be shown in a similar way. □

**Lemma 2.**  $S_{[u]} \supseteq (\Sigma_u, \prec_0, \sqsubset_0)$ .

In this proof, we will include subscripts for equivalence classes to avoid confusing the elements from quotient set  $\Sigma_u / \equiv_{\sqsubset_0}$  with the elements from the quotient comtrace monoid  $\mathbb{S}^* / \equiv_{\theta}$ . In other words, we write  $[\alpha]_{\equiv_{\sqsubset_0}}$  to denote an element of the quotient set  $\Sigma_u / \equiv_{\sqsubset_0}$ , and write  $[u]_{\theta}$  to denote the comtrace generated by  $u$ .

*Proof (of Lemma 2).* Let  $S' = (\Sigma_u, \prec_0, \sqsubset_0)$ . To show  $S_{[u]} \supseteq S'$ , by Proposition 8, it suffices to show  $ext(S_{[u]}) \subseteq ext(S')$ . From Theorem 2, we know that  $ext(S_{[u]_{\theta}}) = \{\triangleleft_w \mid w \in [u]_{\theta}\}$ . Thus we only need to show that for all  $w \in [u]_{\theta}$ ,  $\triangleleft_w \in ext(S')$ .

We observe that from  $u$ , by Definition 4, we can generate all the step sequences in the comtrace  $[u]_{\theta}$  in stages using the following recursive definition:

$$D^0(u) \triangleq \{u\}$$

$$D^n(u) \triangleq \{w \mid w \in D^{n-1}(u) \vee \exists v \in D^{n-1}(u), (v \approx_{\theta} w \vee v \approx_{\theta}^{-1} w)\}$$

Since the set  $[u]_{\theta}$  is finite,  $[u]_{\theta} = D^n(u)$  for some stage  $n \geq 0$ . For the rest of the proof, we will prove by induction on  $n$  that for all  $n \in \mathbb{N}$ , if  $w \in D^n(u)$  then  $\triangleleft_w \in ext(S)$ .

**Base case:** When  $n = 0$ ,  $D^0(u) = \{u\}$ . Since  $\triangleleft_0 \in ext(T)$ , it follows from Proposition 7 that  $\triangleleft_u \in ext(S')$ .

**Inductive case:** When  $n > 0$ , let  $w$  be an element of  $D^n(u)$ . Then either  $w \in D^{n-1}(u)$  or  $w \in (D^n(u) \setminus D^{n-1}(u))$ . For the former case, by inductive hypothesis,  $\triangleleft_w \in ext(S')$ . For the latter case, there must be some element  $v \in D^{n-1}(u)$  such that  $v \approx_{\theta} w$  or  $v \approx_{\theta}^{-1} w$ . By induction hypothesis, we already know  $\triangleleft_v \in ext(S')$ . We want to show that  $\triangleleft_w \in ext(S')$ . There are two cases to consider:

**Case (i)**

When  $v \approx_{\theta} w$ , by Definition 4, there are some  $y, z \in E_{\theta}^*$  and steps  $A, B, C \in \mathbb{S}$  such that  $v = yAz$  and  $w = yBCz$  where  $A, B, C$  satisfy  $B \cap C = \emptyset$  and  $B \cup C = A$  and  $B \times C \subseteq ser$ . Let  $\bar{v} = \bar{y}\bar{A}\bar{z}$  and  $\bar{w} = \bar{y}\bar{B}\bar{C}\bar{z}$  be enumerated step sequences of  $v$  and  $w$  respectively.

Suppose for a contradiction that  $\triangleleft_w \notin ext(S')$ . By Definition 2, there are  $\alpha \in \bar{C}$  and  $\beta \in \bar{B}$  such that  $\alpha \sqsubset_0 \beta$ . We now consider the quotient set  $\bar{A} / \equiv_{\sqsubset_0}$ . By Proposition 4 (1),  $\bar{A} / \equiv_{\sqsubset_0} \subseteq \Sigma_u / \equiv_{\sqsubset_0}$ . Since  $\alpha \sqsubset_0 \beta$ , it follows that  $[\alpha]_{\equiv_{\sqsubset_0}} \hat{\sqsubset}_0 [\beta]_{\equiv_{\sqsubset_0}}$ . Thus, from the fact that  $\hat{\sqsubset}_0$  is partial order, there must exist a chain

$$[\alpha]_{\equiv_{\sqsubset_0}} = [\gamma_1]_{\equiv_{\sqsubset_0}} \hat{\sqsubset}_0^{\text{cov}} [\gamma_2]_{\equiv_{\sqsubset_0}} \hat{\sqsubset}_0^{\text{cov}} \dots \hat{\sqsubset}_0^{\text{cov}} [\gamma_k]_{\equiv_{\sqsubset_0}} = [\beta]_{\equiv_{\sqsubset_0}} \quad (5.1)$$

Then by Theorem 1 and the fact that  $\triangleleft_v \in ext(S')$ , we know that  $\gamma_i \in \bar{A}$  for all  $i$ . In other words, since the chain (5.1) implies that every  $\gamma_i$  must always occur between  $\alpha$  and  $\beta$  in

all stratified extensions of  $S'$  and  $\alpha, \beta \in \bar{A}$ , we also have  $\gamma_i \in \bar{A}$ . Hence, by Proposition 4 (1), we have  $[\gamma_i]_{\equiv_{c_0}} \subseteq \bar{A}$  for all  $i$ ,  $1 \leq i \leq k$ . Also from LC3 of Definition 10 and that  $B \times C \subseteq \text{ser}$ , we know that for each  $\gamma_i$ , either  $[\gamma_i]_{\equiv_{c_0}} \subseteq \bar{B}$  or  $[\gamma_i]_{\equiv_{c_0}} \subseteq \bar{C}$ . Now we note that the first element on the chain  $[\gamma_1]_{\equiv_{c_0}} = [\alpha]_{\equiv_{c_0}} \subseteq \bar{C}$  and the last element on the chain  $[\gamma_k]_{\equiv_{c_0}} = [\beta]_{\equiv_{c_0}} \subseteq \bar{B}$ . Thus, there exist two consecutive elements  $[\gamma_i]_{\equiv_{c_0}}$  and  $[\gamma_{i+1}]_{\equiv_{c_0}}$  on the chain such that  $[\gamma_i]_{\equiv_{c_0}} \subseteq \bar{C}$  and  $[\gamma_{i+1}]_{\equiv_{c_0}} \subseteq \bar{B}$ . But then it follows that

- (a)  $[\gamma_{i+1}]_{\equiv_{c_0}} \times [\gamma_i]_{\equiv_{c_0}} \subseteq \text{ser}$  and  $[\gamma_i]_{\equiv_{c_0}} \hat{\sqsubset}_0^{\text{cov}} [\gamma_{i+1}]_{\equiv_{c_0}}$
- (b)  $\neg([\gamma_i]_{\equiv_{c_0}} \hat{\succ}_0 [\gamma_{i+1}]_{\equiv_{c_0}})$  since  $\triangleleft_v \in \text{ext}(S')$  and  $\gamma_i \frown_{\triangleleft_v} \gamma_{i+1}$

These contradict LC2 of Definition 10 since  $T_0$  is a lso-comtrace.

### Case (ii)

When  $v \approx_{\theta}^{-1} w$ , by Definition 4 there are some  $y, z \in E_{\theta}^*$  and steps  $A, B, C \in \mathbb{S}$  such that  $v = yBCz$  and  $w = yAz$  where  $A, B, C$  satisfy  $B \cap C = \emptyset$  and  $B \cup C = A$  and  $B \times C \subseteq \text{ser}$ . Let  $\bar{v} = \bar{y}\bar{B}\bar{C}\bar{z}$  and  $\bar{w} = \bar{y}\bar{A}\bar{z}$  be enumerated step sequences of  $v$  and  $w$  respectively.

Suppose for a contradiction that  $\triangleleft_w \notin \text{ext}(S')$ . By Definition 2 there are  $\alpha \in \bar{B}$  and  $\beta \in \bar{C}$  such that  $\alpha \prec_0 \beta$ . By Proposition 4 (1),  $\bar{A}/\equiv_{c_0} \subseteq \Sigma_u/\equiv_{c_0}$ . Thus, using a dual argument to the proof of Case (i), we can build a chain

$$[\alpha]_{\equiv_{c_0}} = [\gamma_1]_{\equiv_{c_0}} \hat{\sqsubset}_0^{\text{cov}} [\gamma_2]_{\equiv_{c_0}} \hat{\sqsubset}_0^{\text{cov}} \dots \hat{\sqsubset}_0^{\text{cov}} [\gamma_k]_{\equiv_{c_0}} = [\beta]_{\equiv_{c_0}} \quad (5.2)$$

We then argue that there are two consecutive elements on the chain such that  $[\gamma_i]_{\equiv_{c_0}} \subseteq \bar{B}$  and  $[\gamma_{i+1}]_{\equiv_{c_0}} \subseteq \bar{C}$ , which implies

- (a)  $[\gamma_i]_{\equiv_{c_0}} \times [\gamma_{i+1}]_{\equiv_{c_0}} \subseteq \text{ser}$  and  $[\gamma_i]_{\equiv_{c_0}} \hat{\sqsubset}_0^{\text{cov}} [\gamma_{i+1}]_{\equiv_{c_0}}$
- (b)  $[\gamma_i]_{\equiv_{c_0}} \hat{\succ}_0 [\gamma_{i+1}]_{\equiv_{c_0}}$  since  $\triangleleft_v \in \text{ext}(S')$  and  $\gamma_i \triangleleft_v \gamma_{i+1}$

These contradict LC1 of Definition 10. □

We also need to show that the labeled so-structure defined from each comtrace is indeed a lso-comtrace. In other words, we need to show the following lemma.

**Lemma 3.** *Let  $\theta = (E, \text{sim}, \text{ser})$  be a comtrace alphabet. Given a step sequence  $u \in \mathbb{S}_{\theta}^*$ , the lp-isomorphic class  $[\Sigma_{[u]}, \prec_{[u]}, \sqsubset_{[u]}, \ell]$  is a lso-comtrace over  $\theta$ . □*

The proof of this lemma is straightforward by checking that  $[\Sigma_{[u]}, \prec_{[u]}, \sqsubset_{[u]}, \ell]$  satisfies all conditions LC1–LC5.

**Definition 12 (representation mappings ct2lct and lct2ct).** *Let  $\theta$  be a comtrace alphabet.*

1. *The mapping  $\text{ct2lct} : \mathbb{S}_{\theta}^*/\equiv_{\theta} \rightarrow \text{LCT}(\theta)$  is defined as*

$$\text{ct2lct}(\mathbf{t}) \triangleq [\Sigma_{\mathbf{t}}, \prec_{\mathbf{t}}, \sqsubset_{\mathbf{t}}, \ell],$$

where the function  $\ell : \Sigma_{\mathbf{t}} \rightarrow E$  is defined in Section 2.2 and  $S_{\mathbf{t}} = (\Sigma_{\mathbf{t}}, \prec_{\mathbf{t}}, \sqsubset_{\mathbf{t}})$  is the so-structure defined by the comtrace  $\mathbf{t}$  from Definition 7

2. *The mapping  $\text{lct2ct} : \text{LCT}(\theta) \rightarrow \mathbb{S}_{\theta}^*/\equiv_{\theta}$  is defined as*

$$\text{lct2ct}((X, \prec, \sqsubset, \lambda)) \triangleq \left\{ \text{map}(\lambda, \Omega_{\triangleleft}) \mid \triangleleft \in \text{ext}((X, \prec, \sqsubset)) \right\}. \quad \blacksquare$$

Intuitively, the mapping  $\text{ct2lct}$  is used to convert a comtrace to Isos-comtrace while the mapping  $\text{lct2ct}$  is used to transform a Isos-comtrace into a comtrace. The fact that  $\text{ct2lct}$  and  $\text{lct2ct}$  are valid representation mappings for  $\mathbb{S}_\theta^*/\equiv_\theta$  and  $\text{LCT}(\theta)$  will be shown in the following theorem.

**Theorem 3 (The 1<sup>st</sup> Representation Theorem).** *Let  $\theta$  be a comtrace alphabet.*

1. For every  $\mathbf{t} \in \mathbb{S}_\theta^*/\equiv_\theta$ ,  $\text{lct2ct} \circ \text{ct2lct}(\mathbf{t}) = \mathbf{t}$ .
2. For every  $T \in \text{LCT}(\theta)$ ,  $\text{ct2lct} \circ \text{lct2ct}(T) = T$ .

*Proof.* **1.** The fact that  $\text{ran}(\text{ct2lct}) \subseteq \text{LCT}(\theta)$  follows from Lemma 3. Now for a given  $\mathbf{t} \in \mathbb{S}_\theta^*/\equiv_\theta$ , we have  $\text{ct2lct}(\mathbf{t}) = (\Sigma_{\mathbf{t}}, \prec_{\mathbf{t}}, \sqsubset_{\mathbf{t}}, \ell)$ . Thus, it follows that

$$\begin{aligned} \text{lct2ct}(\text{ct2lct}(t)) &= \{ \text{map}(\ell, \Omega_{\triangleleft}) \mid \triangleleft \in \text{ext}(S_{\mathbf{t}}) \} \\ &= \{ \text{map}(\ell, \Omega_{\triangleleft}) \mid \triangleleft \in \{ \triangleleft_s \mid s \in \mathbf{t} \} \} && \langle \text{by Theorem 2} \rangle \\ &= \{ \text{map}(\ell, \Omega_{\triangleleft_s}) \mid s \in \mathbf{t} \} = \mathbf{t} \end{aligned}$$

**2.** Assume  $T_0 = (\Sigma, \prec_0, \sqsubset_0, \ell)$  is the canonical representation of  $T$ . Observe that since  $T_0 \cong T$ , we have  $\{ \text{map}(\ell, \Omega_{\triangleleft}) \mid \triangleleft \in \text{ext}(T_0) \} = \{ \text{map}(\lambda, \Omega_{\triangleleft}) \mid \triangleleft \in \text{ext}(T) \}$ .

Let  $\Delta = \{ \text{map}(\ell, \Omega_{\triangleleft}) \mid \triangleleft \in \text{ext}(T_0) \}$ . We will next show that  $\Delta \in \mathbb{S}_\theta^*/\equiv_\theta$  and  $\text{ct2lct}(\Delta) = [T_0]$ . Fix an arbitrary  $u \in \Delta$ , from Lemmas 1 and 2,  $S_{[u]} = (\Sigma, \prec_0, \sqsubset_0)$ . From Theorem 2,  $\Delta = \{ \text{map}(\ell, \Omega_{\triangleleft}) \mid \triangleleft \in \text{ext}(S_{[u]}) \} = [u]$ . And the rest follows.  $\square$

The theorem says that the mappings  $\text{ct2lct}$  and  $\text{lct2ct}$  are inverses of each other and hence are both *bijective*.

## 5.2 Representation Theorem for Isos-Comtraces and Combined Dependency Graphs

Using Theorem 3, we are going to show that the *combined dependency graph* notion proposed in [26] is another correct alternative definition for comtraces. First we need to define several representation mappings that are needed for our proofs.

**Definition 13 (representation mappings  $\text{ct2dep}$ ,  $\text{dep2lct}$  and  $\text{lct2dep}$ ).** *Let  $\theta$  be a comtrace alphabet.*

1. The mapping  $\text{ct2dep} : \mathbb{S}_\theta^*/\equiv_\theta \rightarrow \text{CDG}(\theta)$  is defined as

$$\text{ct2dep}(\mathbf{t}) \triangleq (\Sigma_{\mathbf{t}}, \prec_u, \sqsubset_u, \ell),$$

where  $u$  is any step sequence in  $\mathbf{t}$  and  $\prec_u$  and  $\sqsubset_u$  are defined as in Definition 5

2. The mapping  $\text{dep2lct} : \text{CDG}(\theta) \rightarrow \text{LCT}(\theta)$  is defined as  $\text{dep2lct}(D) \triangleq D^\diamond$ .
3. The mapping  $\text{lct2dep} : \text{LCT}(\theta) \rightarrow \text{CDG}(\theta)$  is defined as

$$\text{lct2dep}(T) \triangleq \text{ct2dep} \circ \text{lct2ct}(T). \quad \blacksquare$$

Before proceeding further, we want to make sure that:

- Lemma 4.**
1.  $\text{dep2lct} : \text{CDG}(\theta) \rightarrow \text{LCT}(\theta)$  is a well-defined function.
  2.  $\text{ct2dep} : \mathbb{S}_\theta^*/\equiv_\theta \rightarrow \text{CDG}(\theta)$  is a well-defined function.

*Proof.* **1.** Given a cd-graph  $D_1 = [X, \longrightarrow_1, \dashrightarrow_1, \lambda] \in \text{CDG}(\theta)$ , let  $T = [X, \prec, \sqsubset, \lambda] = D_1^\diamond$ . We know that  $T$  is uniquely defined, since by Definition [11](#)  $(X, \prec, \sqsubset)$  is a so-structure, and so-structures are fixed points of  $\diamond$ -closure (by Proposition [2](#)(4)). We will next show that  $T$  is a lso-comtrace by verifying the conditions LC1–LC5 of Definition [10](#). Conditions LC4 and LC5 are exactly CD1 and CD2.

**LC1:** Suppose for contradiction that there exist two distinct non-serializable sets  $[\alpha]$ ,  $[\beta] \subset X$  such that  $[\alpha](\hat{\leftarrow}^{\text{cov}} \cap \hat{\leftarrow})[\beta]$  and  $\lambda([\alpha]) \times \lambda([\beta]) \subseteq \text{ser}$ . Clearly, this implies that  $\alpha \prec \beta$ , and thus by the  $\diamond$ -closure definition,  $\beta$  is reachable from  $\alpha$  on the directed graph  $G = (X, \rightsquigarrow)$ , where  $\rightsquigarrow = \longrightarrow \cup \dashrightarrow$ . Now we consider a shortest path  $P$

$$\alpha = \delta_1 \rightsquigarrow \delta_2 \rightsquigarrow \dots \rightsquigarrow \delta_{k-1} \rightsquigarrow \delta_k = \beta$$

on  $G$  that connects  $\alpha$  to  $\beta$ . We will prove by induction on  $k \geq 2$  that there exist two consecutive  $\delta_i$  and  $\delta_{i+1}$  on  $P$  such that  $\delta_i \in [\alpha]$  and  $\delta_{i+1} \in [\beta]$  and  $(\lambda(\delta_i), \lambda(\delta_{i+1})) \notin \text{ser}$ , which contradicts with  $\lambda([\alpha]) \times \lambda([\beta]) \subseteq \text{ser}$ .

**Base case:** when  $k = 2$ , then  $\alpha \rightsquigarrow \beta$ . Since  $[\alpha](\hat{\leftarrow}^{\text{cov}} \cap \hat{\leftarrow})[\beta]$ , we have  $\alpha \longrightarrow \beta$ , which by CD3 implies  $(\lambda(\alpha), \lambda(\beta)) \notin \text{ser}$ .

**Inductive case:** when  $k > 2$ , we consider  $\delta_1$  and  $\delta_2$ . If  $\delta_1 \in [\alpha]$  and  $\delta_2 \in [\beta]$ , then by  $[\alpha](\hat{\leftarrow}^{\text{cov}} \cap \hat{\leftarrow})[\beta]$ , we have  $\delta_1 \longrightarrow \delta_2$ , which immediately yields  $(\lambda(\delta_1), \lambda(\delta_2)) \notin \text{ser}$ . Otherwise, we have  $\delta_2 \notin [\alpha] \cup [\beta]$  or  $\{\delta_1, \delta_2\} \subseteq [\alpha]$ . For the first case, we get  $[\alpha]\hat{\leftarrow}[\delta_2]\hat{\leftarrow}[\beta]$ , which contradicts that  $[\alpha]\hat{\leftarrow}^{\text{cov}}[\beta]$ . For the latter case, we can apply induction hypothesis on the path  $\delta_2 \rightsquigarrow \dots \rightsquigarrow \delta_{k-1} \rightsquigarrow \delta_k$ .

LC2 and LC3 can also be shown similarly using “shortest path” argument as above. These proofs are easier since we only need to consider paths with edges in  $\dashrightarrow$ .

**2.** By the proof of [[18](#) Lemma 4.7], for any two step sequences  $t$  and  $u$  in  $\mathbb{S}_\theta^*$ , we have  $u \equiv t$  iff  $\text{ct2dep}([u]) = \text{ct2dep}([t])$ . Thus the mapping  $\text{ct2dep}$  is well-defined.  $\square$

**Lemma 5.** *The mapping  $\text{dep2lct} : \text{CDG}(\theta) \rightarrow \text{LCT}(\theta)$  is injective.*

*Proof.* Assume that  $D_1, D_2 \in \text{CDG}(\theta)$ , such that  $\text{dep2lct}(D_1) = \text{dep2lct}(D_2) = T = [X, \prec, \sqsubset, \lambda]$ . Since  $\diamond$ -closure operator does not change the labeling function, we can assume that  $D_i = [X, \longrightarrow_i, \dashrightarrow_i, \lambda]$  and  $(X, \longrightarrow_i, \dashrightarrow_i)^\diamond = (X, \prec, \sqsubset)$ . We will next show that  $(X, \longrightarrow_1, \dashrightarrow_1) \subseteq (X, \longrightarrow_2, \dashrightarrow_2)$ .

$(\longrightarrow_1 \subseteq \longrightarrow_2)$ : Let  $\alpha, \beta \in X$  such that  $\alpha \longrightarrow_1 \beta$ . Suppose for a contradiction that  $\neg(\alpha \longrightarrow_2 \beta)$ . Since  $\alpha \longrightarrow_1 \beta$ , by CD3,  $(\lambda(\alpha), \lambda(\beta)) \notin \text{ser}$ . Thus, by CD2,  $\beta \dashrightarrow_2 \alpha$ . But since  $(X, \longrightarrow_i, \dashrightarrow_i)^\diamond = (X, \prec, \sqsubset)$ , it follows that  $(X, \longrightarrow_i, \dashrightarrow_i) \subseteq (X, \prec, \sqsubset)$  (by Proposition [2](#)). Thus,  $\alpha \prec \beta$  and  $\beta \sqsubset \alpha$ , a contradiction.

$(\dashrightarrow_1 \subseteq \dashrightarrow_2)$ : Can be proved similarly.

By reversing the role of  $D_1$  and  $D_2$ , we have  $(X, \longrightarrow_1, \dashrightarrow_1) \supseteq (X, \longrightarrow_2, \dashrightarrow_2)$ . Thus, we conclude  $D_1 = D_2$ .  $\square$

We are now ready to show the following representation theorem which ensures that  $\text{lct2dep}$  and  $\text{dep2lct}$  are valid representation mappings for  $\text{LCT}(\theta)$  and  $\text{CDG}(\theta)$ .



**Theorem 4 (The 2<sup>nd</sup> Representation Theorem).** *Let  $\theta$  be a comtrace alphabet.*

1. *For every  $D \in \text{CDG}(\theta)$ ,  $\text{lct2dep} \circ \text{dep2lct}(D) = D$ .*
2. *For every  $T \in \text{LCT}(\theta)$ ,  $\text{dep2lct} \circ \text{lct2dep}(T) = T$ .*

*Proof.* **1.** Let  $D \in \text{CDG}(\theta)$  and let  $T = \text{dep2lct}(D)$ . Suppose for a contradiction that  $E = \text{lct2dep} \circ \text{dep2lct}(D)$  and  $E \neq D$ . From how  $\text{ct2lct}$  is defined,  $\text{ct2lct} = \text{dep2lct} \circ \text{ct2dep}$ . Thus, it follows that  $\text{dep2lct}(E) = T = \text{dep2lct}(D)$ . But this contradicts the injectivity of  $\text{dep2lct}$  from Lemma 5.

**2.** Let  $T \in \text{LCT}(\theta)$  and let  $D = \text{lct2dep}(T)$ . Suppose for a contradiction that  $Q = \text{dep2lct} \circ \text{lct2dep}(T)$  and  $Q \neq T$ . Since  $\text{lct2dep} = \text{ct2dep} \circ \text{lct2ct}$ , if we let  $\mathbf{t} = \text{lct2ct}(T)$ , then  $Q = \text{dep2lct} \circ \text{ct2dep}(\mathbf{t}) \neq T$ . Thus, we have shown that  $\mathbf{t} = \text{lct2ct}(T)$  and  $\text{ct2lct}(\mathbf{t}) = \text{dep2lct} \circ \text{ct2dep}(\mathbf{t}) \neq T$ , contradicting Theorem 3 (2).  $\square$

This theorem shows that Isos-comtraces and cd-graphs are equivalent representations for comtraces. The main advantage of cd-graph definition is its simplicity while the Isos-comtrace definition is stronger and more convenient to prove properties about labeled so-structures that represent comtraces.

We do not need to prove another representation theorem for cd-graphs and comtraces since their representation mappings are simply the composition of the representation mappings from Theorems 3 and 4.

## 6 Composition Operators

Recall for a comtrace monoids  $(\mathbb{S}^* / \equiv_{\theta}, \otimes, [\varepsilon])$ , the comtrace operator  $\_ \otimes \_$  is defined as  $[r] \otimes [t] = [r * t]$ . We will construct analogous composition operators for Isos-comtraces and cd-graphs. We will then show that Isos-comtraces (cd-graphs) over a comtrace alphabet  $\theta$  together with its composition operator form a monoid isomorphic to the comtrace monoid  $(\mathbb{S}^* / \equiv_{\theta}, \otimes, [\varepsilon])$ .

Given two sets  $X_1$  and  $X_2$ , we write  $X_1 \uplus X_2$  to denote the *disjoint union* of  $X_1$  and  $X_2$ . Such disjoint union can be easily obtained by renaming the elements in  $X_1$  and  $X_2$  so that  $X_1 \cap X_2 = \emptyset$ . We define the Isos-comtrace composition operator as follows.

**Definition 14 (composition of Isos-comtraces).** *Let  $T_1$  and  $T_2$  be Isos-comtraces over an alphabet  $\theta = (E, \text{sim}, \text{ser})$ , where  $T_i = [X_i, \prec_i, \sqsubset_i, \lambda_i]$ . The composition  $T_1 \odot T_2$  of  $T_1$  and  $T_2$  is defined as (a lp-isomorphic class of) a labeled so-structure  $[X, \prec, \sqsubset, \lambda]$  such that  $X = X_1 \uplus X_2$ ,  $\lambda = \lambda_1 \cup \lambda_2$ , and  $(X, \prec, \sqsubset) = (X, \prec_{(1,2)}, \sqsubset_{(1,2)})^{\diamond}$ , where*

$$\begin{aligned} \prec_{(1,2)} &= \prec_1 \cup \prec_2 \cup \{(\alpha, \beta) \in X_1 \times X_2 \mid (\lambda(\alpha), \lambda(\beta)) \notin \text{ser}\} \\ \sqsubset_{(1,2)} &= \sqsubset_1 \cup \sqsubset_2 \cup \{(\alpha, \beta) \in X_1 \times X_2 \mid (\lambda(\beta), \lambda(\alpha)) \notin \text{ser}\} \end{aligned} \quad \blacksquare$$

Observe that the operator is well-defined since we can easily check that:

**Proposition 9.** *For every  $T_1, T_2 \in \text{LCT}(\theta)$ ,  $T_1 \odot T_2 \in \text{LCT}(\theta)$ .*  $\square$

We will next show that this composition operator  $\_ \odot \_$  properly corresponds to the operator  $\_ \otimes \_$  of the comtrace monoid over  $\theta$ .

**Proposition 10.** *Let  $\theta$  be a comtrace alphabet. Then*

1. *For every  $R, T \in \text{LCT}(\theta)$ ,  $\text{lct2ct}(R \odot T) = \text{lct2ct}(R) \otimes \text{lct2ct}(T)$ .*
2. *For every  $\mathbf{r}, \mathbf{t} \in \mathbb{S}_\theta^*/\equiv_\theta$ ,  $\text{ct2lct}(\mathbf{r} \otimes \mathbf{t}) = \text{ct2lct}(\mathbf{r}) \odot \text{ct2lct}(\mathbf{t})$ .*

*Proof.* **1.** Assume  $R = [X_1, \prec_1, \sqsubset_1, \lambda_1]$ ,  $T = [X_2, \prec_2, \sqsubset_2, \lambda_2]$  and  $Q = [X_1 \uplus X_2, \prec, \sqsubset, \lambda]$ . We can pick  $\triangleleft_1 \in \text{ext}(R)$  and  $\triangleleft_2 \in \text{ext}(T)$ . Then observe that a stratified order  $\triangleleft$  satisfying  $\Omega_\triangleleft = \Omega_{\triangleleft_1} * \Omega_{\triangleleft_2}$  is an extension of  $Q$ . Thus, by Theorem 3 we have  $\text{lct2ct}(R) \otimes \text{lct2ct}(T) = [\text{map}(\lambda_1, \triangleleft_1)] \otimes [\text{map}(\lambda_2, \triangleleft_2)] = [\text{map}(\lambda, \triangleleft)] = \text{lct2ct}(Q)$  as desired.

**2.** Without loss of generality, we can assume that  $\mathbf{r} = [r]$ ,  $\mathbf{t} = [t]$  and  $\mathbf{q} = [q] = \mathbf{r} \otimes \mathbf{t}$ , where  $q = r * t$ . By appropriate reindexing, we can also assume that  $\Sigma_{\mathbf{q}} = \Sigma_{\mathbf{r}} \uplus \Sigma_{\mathbf{t}}$ . Under these assumptions, let  $\text{ct2lct}(\mathbf{r}) = T_1 = [\Sigma_{\mathbf{r}}, \prec_{\mathbf{r}}, \sqsubset_{\mathbf{r}}, l_1]$ ,  $\text{ct2lct}(\mathbf{t}) = T_2 = [\Sigma_{\mathbf{t}}, \prec_{\mathbf{t}}, \sqsubset_{\mathbf{t}}, l_2]$  and  $\text{ct2lct}(\mathbf{q}) = T = [\Sigma_{\mathbf{q}}, \prec_{\mathbf{q}}, \sqsubset_{\mathbf{q}}, l]$ , where  $l = l_1 \cup l_2$  is simply the standard labeling functions. It will now suffice to show that  $T_1 \odot T_2 = T$ .

$$\begin{aligned} (\subseteq): \text{ Let } T_1 \odot T_2 &= (\Sigma_{\mathbf{r}} \uplus \Sigma_{\mathbf{t}}, \prec_{\langle \mathbf{r}, \mathbf{t} \rangle}, \sqsubset_{\langle \mathbf{r}, \mathbf{t} \rangle}, l)^\diamond. \text{ By Definitions 5 and 7 we have} \\ \prec_{\langle \mathbf{r}, \mathbf{t} \rangle} &= \prec_{\mathbf{r}} \cup \prec_{\mathbf{t}} \cup \{(\alpha, \beta) \in \Sigma_{\mathbf{r}} \times \Sigma_{\mathbf{t}} \mid (\lambda(\alpha), \lambda(\beta)) \notin \text{ser}\} \subseteq \prec_{\mathbf{q}} \\ \sqsubset_{\langle \mathbf{r}, \mathbf{t} \rangle} &= \sqsubset_{\mathbf{r}} \cup \sqsubset_{\mathbf{t}} \cup \{(\alpha, \beta) \in \Sigma_{\mathbf{r}} \times \Sigma_{\mathbf{t}} \mid (\lambda(\beta), \lambda(\alpha)) \notin \text{ser}\} \subseteq \sqsubset_{\mathbf{q}} \end{aligned}$$

Thus, by Proposition 2 (5), we have  $(\Sigma_{\mathbf{r}} \uplus \Sigma_{\mathbf{t}}, \prec_{\langle \mathbf{r}, \mathbf{t} \rangle}, \sqsubset_{\langle \mathbf{r}, \mathbf{t} \rangle}, l)^\diamond \subseteq (\Sigma_{\mathbf{q}}, \prec_{\mathbf{q}}, \sqsubset_{\mathbf{q}}, l)$  as desired. Furthermore, by Proposition 2 (5),  $(\Sigma_{\mathbf{r}} \uplus \Sigma_{\mathbf{t}}, \prec_{\langle \mathbf{r}, \mathbf{t} \rangle}, \sqsubset_{\langle \mathbf{r}, \mathbf{t} \rangle})^\diamond$  is a so-structure.

( $\supseteq$ ): By Definitions 5 and 7 we have  $\prec_{\mathbf{q}} \subseteq \prec_{\langle \mathbf{r}, \mathbf{t} \rangle}$  and  $\sqsubset_{\mathbf{q}} \subseteq \sqsubset_{\langle \mathbf{r}, \mathbf{t} \rangle}$ . Since we already know  $(\Sigma_{\mathbf{r}} \uplus \Sigma_{\mathbf{t}}, \prec_{\langle \mathbf{r}, \mathbf{t} \rangle}, \sqsubset_{\langle \mathbf{r}, \mathbf{t} \rangle})^\diamond$  is a so-structure, it follows from Proposition 2 (5) that

$$(\Sigma_{\mathbf{q}}, \prec_{\mathbf{q}}, \sqsubset_{\mathbf{q}}, l) = (\Sigma_{\mathbf{q}}, \prec_{\mathbf{q}}, \sqsubset_{\mathbf{q}}, l)^\diamond \subseteq (\Sigma_{\mathbf{r}} \uplus \Sigma_{\mathbf{t}}, \prec_{\langle \mathbf{r}, \mathbf{t} \rangle}, \sqsubset_{\langle \mathbf{r}, \mathbf{t} \rangle}, l)^\diamond = T_1 \odot T_2. \quad \square$$

Let  $\mathbb{I}$  denote the lp-isomorphic class  $[\emptyset, \emptyset, \emptyset, \emptyset]$ . Then we observe that  $\text{ct2lct}([\varepsilon]) = \mathbb{I}$  and  $\text{lct2ct}(\mathbb{I}) = [\varepsilon]$ . By Proposition 10 and Theorem 3 the structure  $(\text{LCT}(\theta), \odot, \mathbb{I})$  is isomorphic to the monoid  $(\mathbb{S}_\theta^*/\equiv_\theta, \otimes, [\varepsilon])$  under the isomorphisms  $\text{ct2lct} : \mathbb{S}_\theta^*/\equiv_\theta \rightarrow \text{LCT}(\theta)$  and  $\text{lct2ct} : \text{LCT}(\theta) \rightarrow \mathbb{S}_\theta^*/\equiv_\theta$ . Thus, the triple  $(\text{LCT}(\theta), \odot, \mathbb{I})$  is also a monoid. We can summarize these facts in the following theorem:

**Theorem 5.** *The mappings  $\text{ct2lct}$  and  $\text{lct2ct}$  are monoid isomorphisms between two monoids  $(\mathbb{S}_\theta^*/\equiv_\theta, \otimes, [\varepsilon])$  and  $(\text{LCT}(\theta), \odot, \mathbb{I})$ .*  $\square$

Similarly, we can also define a composition operator for cd-graphs.

**Definition 15 (composition of cd-graphs).** *Let  $D_1$  and  $D_2$  be cd-graphs over an alphabet  $\theta = (E, \text{sim}, \text{ser})$ , where  $D_i = [X_i, \longrightarrow_i, \dashrightarrow_i, \lambda_i]$ . The composition  $D_1 \odot D_2$  of  $D_1$  and  $D_2$  is defined as (a lp-isomorphic class of) a labeled so-structure  $[X, \longrightarrow, \dashrightarrow, \lambda]$  such that  $X = X_1 \uplus X_2$ ,  $\lambda = \lambda_1 \cup \lambda_2$ , and*

$$\begin{aligned} \longrightarrow &= \longrightarrow_1 \cup \longrightarrow_2 \cup \{(\alpha, \beta) \in X_1 \times X_2 \mid (\lambda(\alpha), \lambda(\beta)) \notin \text{ser}\} \\ \dashrightarrow &= \dashrightarrow_1 \cup \dashrightarrow_2 \cup \{(\alpha, \beta) \in X_1 \times X_2 \mid (\lambda(\beta), \lambda(\alpha)) \notin \text{ser}\} \quad \blacksquare \end{aligned}$$

From this definition, it is straightforward to show the following propositions, which we will state without proofs.

**Proposition 11.** *For every  $D_1, D_2 \in \text{CDG}(\theta)$ ,  $D_1 \odot D_2 \in \text{CDG}(\theta)$ .* □

**Proposition 12.** *Let  $\theta$  be a comtrace alphabet. Then*

1. *For every  $R, T \in \text{LCT}(\theta)$ ,  $\text{lct2dep}(R \odot T) = \text{lct2dep}(R) \odot \text{lct2dep}(T)$ .*
2. *For every  $D, E \in \text{CDG}(\theta)$ ,  $\text{dep2lct}(D \odot E) = \text{dep2lct}(D) \odot \text{dep2lct}(E)$ .* □

Putting the two preceding propositions and Theorem 4 together, we conclude:

**Theorem 6.** *The mappings  $\text{lct2dep}$  and  $\text{dep2lct}$  are monoid isomorphisms between two monoids  $(\text{LCT}(\theta), \odot, \mathbb{I})$  and  $(\text{CDG}(\theta), \odot, \mathbb{I})$ .* □

## 7 Conclusion

The simple yet useful construction we used extensively in this paper is to build a quotient so-structure modulo the  $\square$ -cycle equivalence relation. Intuitively, each  $\square$ -cycle equivalence class consists of all the events that must be executed simultaneously with one another and hence can be seen as a single “composite event”. The resulting quotient so-structure is technically easier to handle since both relations of the quotient so-structure are acyclic. From this construction, we were able to give a labeled so-structure definition for comtraces similar to the labeled poset definition for traces. This quotient construction also explicitly reveals the following connection: a step on a step sequence  $s$  is not serializable with respect to the relation  $ser$  of a comtrace alphabet if and only if it corresponds to a  $\square$ -cycle equivalence class of the Isos-comtrace representing the comtrace  $[s]$  (cf. Proposition 4).

We have also formally shown that the quotient monoid of comtraces, the monoid of Isos-comtraces and the monoid of cd-graphs *over the same comtrace alphabet* are indeed isomorphic by establishing monoid isomorphisms between them. These three models are formal linguistic, order-theoretic, and graph-theoretic respectively, which allows us to apply a variety of tools and techniques.

An immediate future task is to develop a framework similar to the one in this paper for *generalized comtraces*, proposed and developed in [21][27][22]. Generalized comtraces extend comtraces with the ability to model events that can be executed *earlier than or later than but never simultaneously*. Another direction is to define and analyze infinite comtraces (and generalized comtraces) in a spirit similar to the works on infinite traces, e.g., [13][4]. It is also promising to use infinite Isos-comtraces and cd-graphs to develop logics for comtraces similarly to what have been done for traces (cf. [31][7]).

**Acknowledgments.** I am grateful to Prof. Ryszard Janicki for introducing me comtrace theory. I also thank the Mathematics Institute of Warsaw University and the Theoretical Computer Science Group of Jagiellonian University for their supports during my visits. It was during these visits that the ideas from this paper emerge. This work is financially supported by the Ontario Graduate Scholarship and the Natural Sciences and Engineering Research Council of Canada. The anonymous referees are thanked for their valuable comments that help improving the readability of this paper.

## References

1. Clarke, E., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (1999)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms*, 2nd edn. MIT Press, Cambridge (2001)
3. Davey, B.A., Priestley, H.A.: *Introduction to Lattices and Order*. Cambridge University Press, Cambridge (2002)
4. Diekert, V.: On the Concatenation of Infinite Traces. In: Jantzen, M., Choffrut, C. (eds.) STACS 1991. LNCS, vol. 480, pp. 105–117. Springer, Heidelberg (1991)
5. Diekert, V., Rozenberg, G. (eds.): *The Book of Traces*. World Scientific, Singapore (1995)
6. Diekert, V., Métivier, Y.: Partial Commutation and Traces. In: *Handbook of Formal Languages, Beyond Words*, vol. 3, pp. 457–533. Springer, Heidelberg (1997)
7. Diekert, V., Horsch, M., Kufleitner, M.: On First-Order Fragments for Mazurkiewicz Traces. *Fundam. Inform.* 80(1-3), 1–29 (2007)
8. Esparza, J., Heljanko, K.: *Unfoldings – A Partial-Order Approach to Model Checking*. Springer, Heidelberg (2008)
9. Farzan, A., Madhusudan, P.: Causal Dataflow Analysis for Concurrent Programs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)
10. Farzan, A., Madhusudan, P.: Causal Atomicity. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 102–116. Springer, Heidelberg (2007)
11. Fishburn, P.C.: *Interval Orders and Interval Graphs*. J. Wiley, New York (1985)
12. Gaifman, H., Pratt, V.: Partial Order Models of Concurrency and the Computation of Function. In: *Proc. of LICS 1987*, pp. 72–85 (1987)
13. Gastin, P.: Infinite Traces. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 277–308. Springer, Heidelberg (1990)
14. Gazagnaire, T., Genest, B., Héluouët, L., Thiagarajan, P.S., Yang, S.: Causal Message Sequence Charts. *Theor. Comput. Sci.* 410(41), 4094–4110 (2009)
15. Janicki, R.: Relational Structures Model of Concurrency. *Acta Informatica* 45(4), 279–320 (2008)
16. Janicki, R., Koutny, M.: Invariants and Paradigms of Concurrency Theory. In: Aarts, E.H.L., van Leeuwen, J., Rem, M. (eds.) PARLE 1991. LNCS, vol. 506, pp. 59–74. Springer, Heidelberg (1991)
17. Janicki, R., Koutny, M.: Structure of Concurrency. *Theoretical Computer Science* 112(1), 5–52 (1993)
18. Janicki, R., Koutny, M.: Semantics of Inhibitor Nets. *Information and Computation* 123(1), 1–16 (1995)
19. Janicki, R., Koutny, M.: Fundamentals of Modelling Concurrency Using Discrete Relational Structures. *Acta Informatica* 34, 367–388 (1997)
20. Janicki, R., Koutny, M.: On Causality Semantics of Nets with Priorities. *Fundamenta Informaticae* 34, 222–255 (1999)
21. Janicki, R., Lê, D.T.M.: Modelling Concurrency with Quotient Monoids. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 251–269. Springer, Heidelberg (2008)
22. Janicki, R., Lê, D.T.M.: Modelling Concurrency with Comtraces and Generalized Comtraces (submitted in 2009), <http://arxiv.org/abs/0907.1722>
23. Juhás, G., Lorenz, R., Mauser, S.: Causal Semantics of Algebraic Petri Nets distinguishing Concurrency and Synchronicity. *Fundamenta Informatica* 86(3), 255–298 (2008)
24. Juhás, G., Lorenz, R., Mauser, S.: Synchronous + Concurrent + Sequential = Earlier Than + Not Later Than. In: *Proc. of ACSD 2006*, Turku, Finland, pp. 261–272. IEEE Press, Los Alamitos (2006)

25. Kleijn, H.C.M., Koutny, M.: Process Semantics of General Inhibitor Nets. *Information and Computation* 190, 18–69 (2004)
26. Kleijn, J., Koutny, M.: Formal Languages and Concurrent Behaviour. *Studies in Computational Intelligence* 113, 125–182 (2008)
27. Lê, D.T.M.: *Studies in Comtrace Monoids*, Master Thesis, Dept. of Computing and Software, McMaster University, Canada (August 2008)
28. Mazurkiewicz, A.: Concurrent Program Schemes and Their Interpretation, TR DAIMI PB-78, Comp. Science Depart., Aarhus University (1977)
29. Pratt, V.: Modeling concurrency with partial orders. *International Journal of Parallel Programming* 15(1), 33–71 (1986)
30. Szpilrajn, E.: Sur l'extension de l'ordre partiel. *Fund. Mathematicae* 16, 386–389 (1930)
31. Thiagarajan, P.S., Walukiewicz, I.: An expressively complete linear time temporal logic for Mazurkiewicz traces. *Inf. Comput.* 179(2), 230–249 (2002)

# Integrated Process Planning and Supply Chain Configuration for Commodity Assemblies Using Petri Nets

Oleg Gusikhin and Erica Klampfl

Ford Research and Advanced Engineering, Dearborn, MI 48124, USA

**Abstract.** We present a methodology for integrated process planning and supply chain configuration for commodity assemblies. Although the supply chain configuration problem for commodity assemblies is relatively straightforward using math programming, developing a commodity-dependent math program with precedence constraints can be a very daunting and time-consuming process. We use Petri net techniques to support the development of such a math program. Our modeling approach is based on a series of stepwise Petri net transformations that transform the Petri net model of the assembly process into a supply chain configuration representation. We use the matrix representation as a basis for the integer program formulation. We present a small example commodity from the automotive industry to illustrate the proposed methodology.

## 1 Introduction

In this paper, we present a methodology for integrated process planning and supply chain design for commodity assemblies. Commodity assemblies are items where the technological operations are relatively simple, primarily manual, and well-known: the jobs can be outsourced to many suppliers or performed in-house. Outsourcing commodity assembly jobs to suppliers typically does not require elaborate know-how or substantial capital investment in tooling and infrastructure. Original Equipment Manufacturer (OEM)s might even use the spot market (i.e., a one time buy, in contrast with a long term contract, of parts or capacity immediately available) for the components and/or capacity. Consequently, cost is the primary consideration for such a supply chain. Designing the supply chain for commodity items is often performed by the OEM, itself, to ensure a minimum cost supply chain. The first question asked in the design process is whether and which jobs should be outsourced or performed in-house: this is referred to as the “make” or “buy” decision. In the case of the “buy” decision for the entire assembly and/or subassemblies, the OEM must decide on the specific supplier in which to source the jobs. Although in the case of the “buy” decision, the design of the upstream supply chain is usually the responsibility of the suppliers; however, it is beneficial for the OEM to have an idea of the potential cost implications to the entire supply chain.

Setting up a supply chain for commodity items is not necessarily a trivial task. Even for relatively small assemblies, the number of possible process alternatives

may be very high. In addition, the number of sourcing alternatives for each operation could be high, as well. This is especially true in the case of emerging economies, such as Asia-Pacific or Latin America, where consideration needs to be given not only to labor and transportation cost, but also to border tariffs, currency fluctuations, and local taxes.

Mathematical programming is commonly used to model the design of distribution networks, where whole units flow through the network [1]. This does not involve the additional product build-up through the network or the representation of precedence constraints (i.e., constraints that define which parts must be assembled before other parts). While it is not necessarily difficult to formulate supply chain choices for a commodity as a math program for a specific part, to generate the math program for any product with varying precedence constraints, potential suppliers, etc. is a much more daunting task, often customized for each individual case. Additionally, while network modeling has been proven to be efficient for representing supply chain problems, classical network models allow for only one type of node that can be used either to represent an assembly or an alternative.

To overcome the math program set-up issues, we present a Petri net representation that provides two types of nodes, allowing us to capture both the assembly and the alternative. Petri nets have been broadly used for both process planning and supply chain configuration problems. Assembly planning has been extensively addressed in the literature, especially relative to automatic robot planning. Rosell summarizes autonomous robotics assembly problems and reviews the approaches that use Petri nets as a formalism to develop the corresponding planners [2]. Petri nets' clear graphical representation of complex assembly relations has led to many applications in manufacturing and supply chain analysis and design. Gusikhin and Kulnitch discuss integrated process planning, routing, and scheduling for flexible manufacturing system using Petri nets [14]. Viswanadham and Ragavhan present modeling techniques for analyzing the supply chain process using generalized stochastic Petri nets: they develop a methodology to compare the performance of make-to-stock and assemble-to-order policies in terms of total cost and to locate supply chain push-pull decoupling points [3]. You et. al. present a set of formalisms based on color Petri-nets to model and evaluate supply chain configurations considering product, process, and logistics design [4]. Li et.al. discuss supply route optimization based on Petri Net models [5]. Zimmerman et. al. describe a variant of the colored stochastic Petri Net model used by General Motors' supply chain to evaluate vehicle order-to delivery time [6].

In this paper, we propose a method to formulate the integrated process planning and design of the supply chain for commodity assemblies problem using Petri Net techniques. The advantage of using Petri Nets is that their graphical nature provides an unambiguous and concise representation of the problem, which is directly transferable to an integer program (IP) representation utilizing the Petri net state equation. In the next section, we present the general problem formulation and discuss terminology. In Section 3, we discuss and illustrate a

Petri Net model for the simple product assembly process and demonstrate how the model development is carried out as a series of stepwise net transformations [9]. Section 3.1 provides an example of Petri net modeling of the assembly process. Section 3.2 describes the transformation rules from a generic assembly process model to a product variation assembly process model. In Section 3.3, we present the transformation from the assembly process to the supply chain model. Section 3.4 outlines the formulation of the IP using the state equation of the supply chain Petri Net model as a basis. We conclude the paper with a summary in Section 4.

## 2 Problem Formulation

In this section, we present the parameter definitions that will be used throughout the paper and provide an overview of the problem. This paper focuses on how to design the supply chain for a product (i.e., commodity). A product is made up of several components, some standard and some with variation. Each component needs to be fabricated at a fabrication supplier, assembled at some point in the supply chain, and the final product shipped to the assembly plants to meet the plants' demands for the different product variations. The problem we address is to find the lowest cost way to assemble and ship the different products from the fabrication suppliers through the supply chain to the assembly plants. This entails a selection of the subset of fabrication and assembly suppliers capable of performing respective operations (subject to assembly precedence constraints) and the assignment of the production volumes at each of the chosen suppliers (subject to supplier capacity limitations). At this stage of the supply chain design for commodity assemblies, the primary consideration is given to manufacturing and transportation costs that are used to select the set of candidate solutions for further analysis.

The first few parameters listed in Table 1 define the components in a final product and the unique final product variants.

**Table 1.** This table contains parameters that define the components in a final product and the unique final product variants

Parameter	Description
$\kappa$	number of component types in a final product
$\mathcal{K} = \{1, \dots, \kappa\}$	set of component types
$\beta$	number of unique components (includes component varieties)
$\mathcal{B} = \{1, \dots, \beta\}$	set of unique components
$\nu$	number of final product variants
$\mathcal{V}_i \subseteq \mathcal{B}$	set that defines the product variant composition for variant $i = 1, \dots, \nu$ such that $ \mathcal{V}_i  = \kappa$
$\mathcal{V} = \{\mathcal{V}_i : i = 1, \dots, \nu\}$	set of the set of product variations (i.e., the finished products)



A typical sized commodity has on the order of twenty components, fifteen product variations, and twenty operations, with a choice of up to six potential suppliers for each assembly operation. Figure 1 shows the engine compartment side module (ECSM), which is a representative commodity example. In this paper, we introduce a smaller example commodity (see the tube assembly in Fig. 2) to use as a running illustration to describe the methodology.

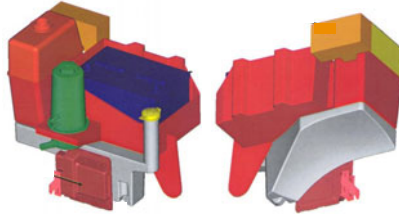


Fig. 1. Example commodity: ECSM

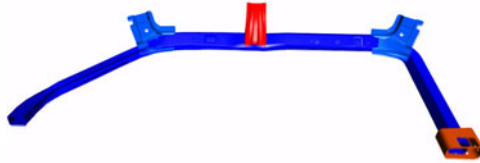


Fig. 2. Example commodity: tube assembly

Figure 3 shows the components that constitute the given tube assembly. There are  $\kappa = 5$  component types: tube surface (tube\_surf), front horn (frt\_horn), mid horn (mid\_horn), rear horn (rear\_horn), and lower coupler (lwr\_cplr). There are two types of lwr\_cplr, which we refer to as L1 and L2 and two types of mid\_horns, which we refer to as MH1 and MH2. Table 2 shows the feasible products and their associated components for the tube assembly: the number of final product variants  $\nu = 3$ . We define our product variant sets as follows:  $\mathcal{V}_1 = \{T, L1, FH, RH, MH1\}$ ,  $\mathcal{V}_2 = \{T, L1, FH, RH, MH2\}$ , and  $\mathcal{V}_3 = \{T, L2, FH, RH, MH2\}$ . Note that  $\mathcal{V}_1$ ,  $\mathcal{V}_2$ , and  $\mathcal{V}_3$  correspond to the columns in Table 2 under A, B, and C, respectively. Hence,  $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3\} = \{\{T, L1, FH, RH, MH1\}, \{T, L1, FH, RH, MH2\}, \{T, L2, FH, RH, MH2\}\}$ .

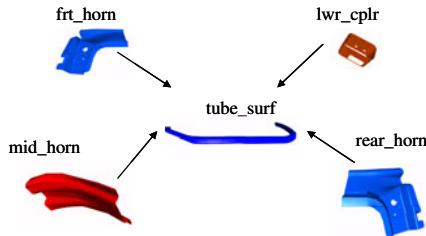


Fig. 3. Five component types in the tube assembly

**Table 2.** Tube assembly product variations

Components	Products		
	A	B	C
tube_surf	T	T	T
lwr_cplr	L1	L1	L2
frt_horn	FH	FH	FH
rear_horn	RH	RH	RH
mid_horn	MH1	MH2	MH2

Next, we introduce the parameters that describe the subassemblies and their relations to the assembly operations in Table 3.

**Table 3.** Parameters describing the subassemblies and their relations to the assembly operations

Parameter	Description
$\eta$	number of valid subassemblies
$\mathcal{H}_i \subseteq \mathcal{B}$	set of unique components that make up a valid subassembly for $i = 1, \dots, \eta$
$\mathcal{H} = \{\mathcal{H}_i : i = 1, \dots, \eta\}$	set of all valid subassemblies
$\theta$	number of valid assembly operations (i.e., satisfy precedence constraints)
$\mathcal{O}_i$	a pair consisting of a subassembly $\mathcal{H}_j \in \mathcal{H}$ and a component $b \in \mathcal{B}$ that together form a new subassembly for $i = 1, \dots, \theta$ . Note that for fabrication operations, the pair consists just of the component and the empty set.
$\mathcal{O} = \{\mathcal{O}_i : i = 1, \dots, \theta\}$	set of fabrication and assembly operations
$\mathcal{O}^f \subseteq \mathcal{O}$	set of fabrication operations
$\gamma$	number of possible assembly sequences
$\mathcal{G}_i \subseteq \mathcal{B}$	set that defines the order of components in a possible assembly sequence, for $i = 1, \dots, \gamma$ such that $ \mathcal{G}_i  = \kappa$
$\mathcal{G} = \{\mathcal{G}_i : i = 1, \dots, \gamma\}$	set of possible assembly sequences

Consider the subassemblies for product “A” in Table 2, which is comprised of five components (T, L1, FH, RM, MH1) and is produced by fabrication operations  $\mathcal{O}_1$  through  $\mathcal{O}_5$ . Note that the components for tube\_surf (T), frt\_horn (FH), rear\_horn (RH), and their corresponding fabrication operations are common for the entire product family, while lwr\_cplr (L1) is used for products “A” and “B,” and mid\_horn (MH1) is unique for product “A.” Figure 4 shows a key for valid subassemblies for product “A.” Note that the component “T” appears in all subassemblies because it has precedence over all other components, no other components have precedence over each other, and the elements within a set are not ordered.

Assembly operations are defined between one subassembly and a component, and fabrication operations consist of production of individual components. For product “A” in Table 2, there are 32 possible assembly operations: the the components for tube\_surf, frt\_horn, and rear\_horn can be assembled to tube\_surf in

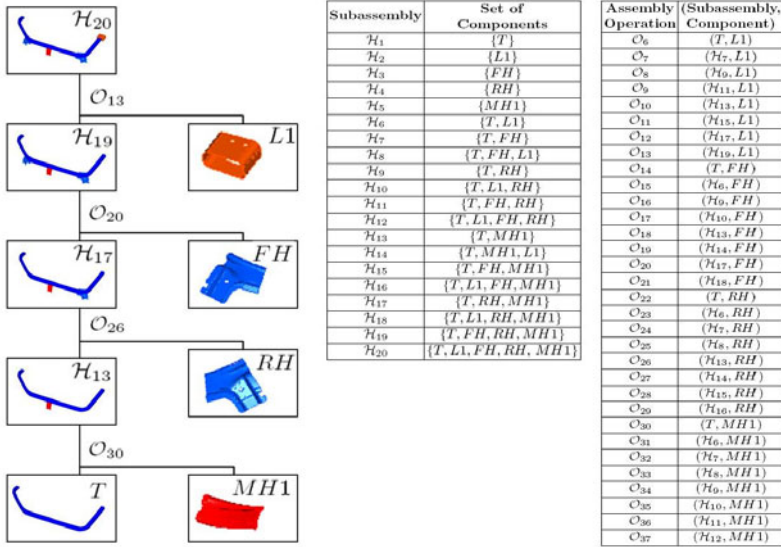


Fig. 4. Example assembly sequence for product “A” and a list of possible assembly operations

Table 4. Parameters associated with facilities

Parameter	Description
$\alpha$	number of assembly plants
$\mathcal{A} = \{1, \dots, \alpha\}$	set of assembly plants: these plants receive the final subassemblies (i.e., products)
$\sigma$	number of fabrication suppliers
$\mathcal{F} = \{1, \dots, \sigma\}$	set of fabrication suppliers: these suppliers make the individual components
$\mathcal{F}_b \subset \mathcal{F}$	set of fabrication suppliers that can make component $b \in \mathcal{B}$
$\xi$	number of assembly suppliers
$\mathcal{S} = \{1, \dots, \xi\}$	set of assembly suppliers: these suppliers assemble the individual components
$\mathcal{S}_o \subset \mathcal{S}$	set of assembly suppliers that can perform assembly operations $o \in \mathcal{O}$
$d_{av}$	demand of assembly plant $a \in \mathcal{A}$ for product $v \in \mathcal{V}$
$\mathcal{D} = \{d_{av}\}$	set of demands for assembly plant $a \in \mathcal{A}$ for product $v \in \mathcal{V}$
$c_{bf}^F$	cost to fabricate component $o \in \mathcal{O}^F$ at fabrication supplier $f \in \mathcal{F}_b$
$c_{os}^A$	cost for assembly operation $o \in \mathcal{O} \setminus \mathcal{O}^F$ at assembly supplier $s \in \mathcal{S}_o$
$c_{zs_1, s_2}^T$	transportation cost to ship $z \in \mathcal{B} \cup \mathcal{H}$ (where $z$ is a unique component or a sub assembly) from supplier $s_1 \in \mathcal{F} \cup \mathcal{S}$ (either a fabrication supplier or an assembly supplier) to $s_2 \in \mathcal{S} \cup \mathcal{A}$ (either an assembly supplier or assembly plant).
$l_{om}$	load factor for operation $o \in \mathcal{O}$ at fabrication and assembly suppliers $m \in \mathcal{F} \cup \mathcal{S}_o$
$q_s$	capacity of fabrication and assembly supplier $m \in \mathcal{F} \cup \mathcal{S}$

any sequence. Figure 4 shows one possible assembly process for product “A” and provides a key for the assembly operations.

For each product variant in Table 2, there are many possible assembly sequences: these are ordered sequences and must each contain  $\kappa = 5$  component types. We provide a few assembly sequences for product variant “A” to illustrate the terminology:

$$\mathcal{G}_1 = \{T, L1, FH, RH, MH1\}, \mathcal{G}_2 = \{T, L1, RH, FH, MH1\}, \mathcal{G}_3 = \{T, L1, MH1, FH, RH\}, \mathcal{G}_4 = \{T, L1, FH, MH1, RH\}, \text{ and } \mathcal{G}_5 = \{T, L1, MH1, RH, FH\}.$$

Note that any possible assembly sequence for a particular product must contain the same components as defined in its product variant. For example,  $\mathcal{G}_1 \cap \mathcal{V}_1 = \emptyset, \dots, \mathcal{G}_5 \cap \mathcal{V}_1 = \emptyset$  because  $\mathcal{G}_1$  through  $\mathcal{G}_5$  are all possible assembly sequences for product variant “A,” and  $\mathcal{V}_1$  is the product variant composition for product variant “A.”

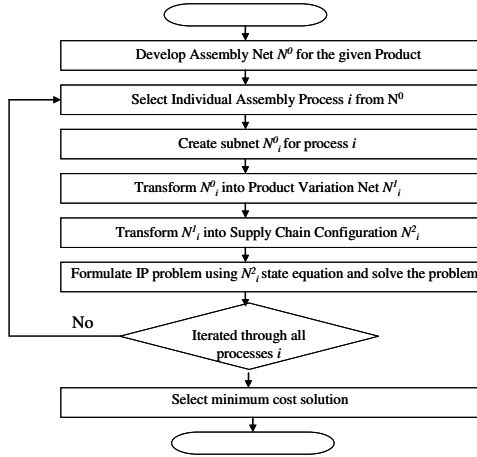
The next several parameters listed in Table 4 are associated with facilities: assembly plants, assembly suppliers, and fabrication suppliers.

### 3 Petri Net Modeling of the Supply Chain Configuration Problem

In this section, we describe the steps in the modeling process that lead to the IP formulation of the supply chain configuration problem. Figure 5 shows the process overview. The process commences with the development of the Petri net model representation of the assembly process for a given product. This model allows us to obtain all possible assembly sequences associated with the set of minimal t-invariant of the assembly net. Next, we iterate over all possible assembly sequences to obtain the subnet of the assembly net that corresponds to the given assembly process. Then, we transform this subnet into a product variation net that represents the assembly process model: this expands the generic assembly sequence for a given product family to specific assembly operations and components. The next step is to transform the product variation net to the supply chain configuration net: this substitutes transitions corresponding to assembly operations and places corresponding to product configuration with operations at specific suppliers and specific products at the supply chain locations augmented by the transitions to represent the logistics infrastructure. Using an algebraic linear representation for this Petri net, we directly formulate the IP, which can be solved using any IP solver, such as IBM CPLEX [12].

#### 3.1 Assembly Process Net

The modeling process starts with designing a model that represents the generic assembly sequence of a given product type. There has been extensive research in automatically obtaining assembly sequences from the geometric relationships between the components of the part derived from the mechanical computer aided design (CAD) of the part. Mello and Sanderson present an algorithm that takes



**Fig. 5.** The modeling process

a representation of the product as input and generates the set of all feasible assembly sequences, which is represented as an AND/OR graph [10]. Cao and Sanderson introduce an AND/OR net as a framework for representation and reasoning about geometric constraints in a robotic work cell system and provide a method for mapping the AND/OR net to a Petri net [11]. Other research has been done on how an assembly Petri net (APN) can be built from a set of precedence constraints or from an AND/OR representation [8, 7]. Furthermore, these authors demonstrate how to use a Petri net state equation to formulate process planning optimization (to minimize disassembly energy or maximize parallelism) as a Linear Programming problem that can be solved using any of the standard methods.

In building our assembly Petri net model, we follow the same assumptions as in [7]:

- Exactly two subassemblies are joined at each assembly task.
- After parts have been put together, they remain together until the end of the assembly process.
- Whenever two parts are joined, all contacts between them are established.
- Assembly operations and disassembly operations are invertible with respect to each other.
- Assembly and disassembly operations do not exist simultaneously in one sequence.

We define the assembly net graph by the tuple  $N = (P, T, W)$ , where

- $P$  is a finite set of places associated with component types or subassemblies of the given product.
- $T = T_f \cup T_a \cup \{t_c\}$  is a set of transitions that represents the manufacturing operations (fabrication, assembly, and product consumption), where

- $T_f$  is a finite set of transitions associated with component fabrication operations: these are source transitions that have exactly one output place;
  - $T_a$  is a finite set of transitions associated with assembly operations: these transitions have exactly two input places and one output place; and
  - $t_c$  is a transition associated with the product consumption operation: this is a sink transition with one input place that represents the final assembly.
- $W : (P \times T) \cup (T \times P) \rightarrow \{0, 1\}$  defines the set of directed arcs in the net.

Consider the tube assembly presented in Fig. 2. This is a very small product consisting of five component types: tube surface, front horn, mid horn, rear horn, and lower coupler. The latter four component types can be assembled to the tube surface in any sequence. We present the resulting assembly net for this example in Fig. 6: the squares with the component names represent the fabrication of the component, the squares with the “t’s” are the transitions representing assembly operations (the square marked tube assembly corresponds to the product consumption operation), and the circles represent components and subassemblies. We generated Fig. 6 and Fig. 7 from a Petri net markup language file using the PN Kernel tool 17.

$M_0 : P \rightarrow \{0, 1\}$  is a marking of the assembly net that represents the current status of the assembly process. Assume that our initial marking of each place is zero. In order to start the assembly process, we assume that we have all individual fabrication components on hand that correspond to the marking that has one token in every output place of the fabrication transitions,  $T_f$ . This marking

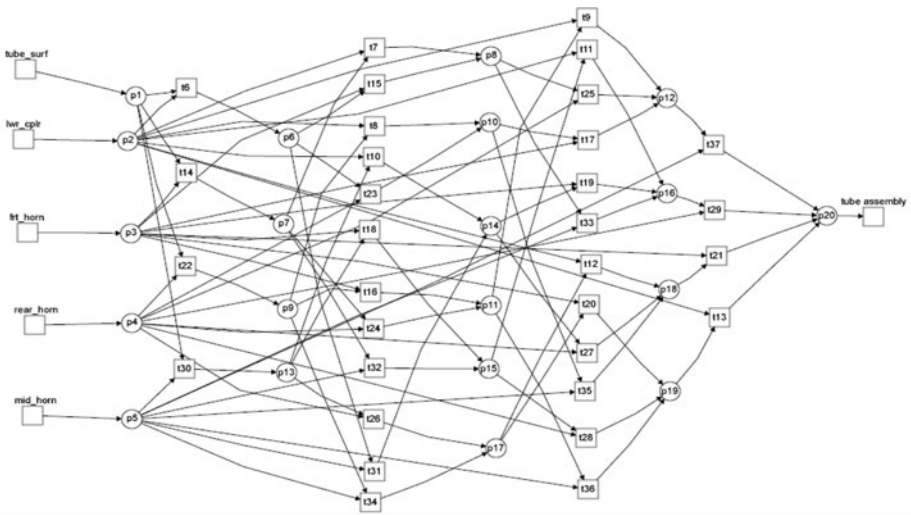


Fig. 6. Assembly Petri net model for the tube assembly

can be obtained by a single firing of transitions from  $T_f$ . At every point in the assembly process, the set of possible next assembly operations is defined by the set of enabled transitions from  $T_a$  (i.e., transitions from  $T_a$  that have tokens in all their input places). When the next assembly operation is selected, the corresponding transition is fired, removing the tokens from its input places and adding a token to its output place. The last operation is the consumption firing,  $t_c$ , which removes the token from the net. Any firing sequence that includes a single firing of every transition from  $T_f$  and a single firing of a transition  $t_c$  corresponds to a valid assembly sequence. This firing sequence is equivalent to the minimal t-invariant of  $N$ . Consequently, all possible assembly sequences can be obtained by generating the set of all minimal t-invariants of  $N$  (see [13] for an overview of methods for obtaining the t-invariant). Figure 7 shows one such t-invariant overlaid on the Petri net model.

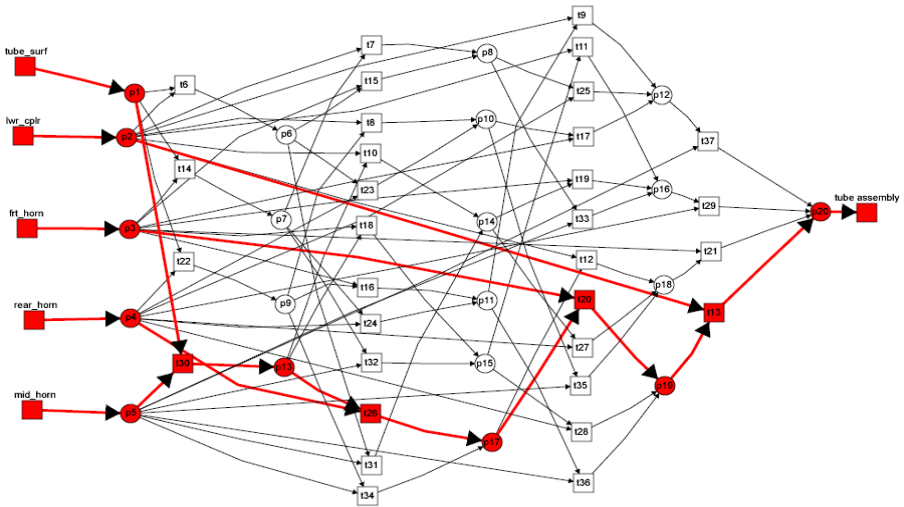


Fig. 7. Example t-invariant for the tube assembly overlaid on the Petri net model

### 3.2 Product Variation Net

The assembly net described in the previous section provides a generic assembly process for a given product. However, in most cases, products may have several variations and/or options for different component types. These variations are defined by the engineering bill of material (EBOM) that lists all the unique components constituting different variations of the same product. Note that the EBOM is different from the manufacturing bill of material (MBOM): the EBOM lists all the specific components that constitute a given variant of the product, and the MBOM captures the specific assembly sequence represented by a multilevel structure where each intermediate level corresponds to the specific subassembly.

One of the requirements of process planning and supply chain configuration is to ensure that the assembly sequence is kept the same across all product variants and supply chain paths. In order to address this, we iterate over each assembly process plan (the set of minimum t-invariants): at each iteration, we formulate and solve the supply chain configuration optimization problem. This guarantees that all product variants and supply chain product paths will follow the same assembly sequence and ensures the single MBOM. We compare the optimal solutions for each individual process and select the best set of solutions for further analysis.

At each iteration  $i$ , we use the t-invariant to create a subnet  $N_i$  from  $N$  that consists of all the transitions constituting the given t-invariant together with their input and output places and connecting arcs. Figure 8 shows an example of the t-invariant based net,  $N_i$ , for the tube assembly.

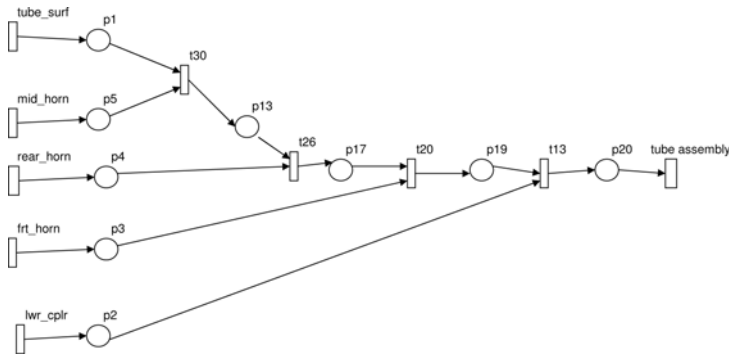


Fig. 8. Example t-invariant based net,  $N_i$ , for the tube assembly

The product variation net graph is defined by the 3-tuple  $N^1 = (P^1, T^1, W^1)$ , where  $P^1$  is a finite set of places that represent specific components or sub-assemblies. Each place is associated with the set  $\mathcal{H}_j$  for  $j \in 1, \dots, \eta$  that defines the unique components comprising the subassembly at a given point in the process.  $T^1$  is a finite set of transitions corresponding to  $\mathcal{O}_j$  for  $j = 1, \dots, \theta$ .  $W^1 : (P^1 \times T^1) \cup (T^1 \times P^1) \rightarrow \{0, 1\}$  defines the set of directed arcs in the net.

The transformation of the subnet  $N_i$  into a product variation net is an iterative process of unfolding the transitions and their output places into a set of transitions and a set of output places associated with the specific assembly operations from  $\mathcal{O}$  and specific variants of the subassembly from  $\mathcal{H}$ . The procedure starts by exploring the source transitions corresponding to component fabrication operations. Each transition is replaced by a set of transitions with output places corresponding to the fabrication operations that produce individual variants of the given component. The output place,  $p_j$ , of the transition  $t_r \in N_i$  is replaced by the set of new places  $P_j^1 \subseteq P^1$  that corresponds to the specific variants of the given component type (see Fig. 9). In order to demonstrate the transformation process from one net to another net, from this point forward, we



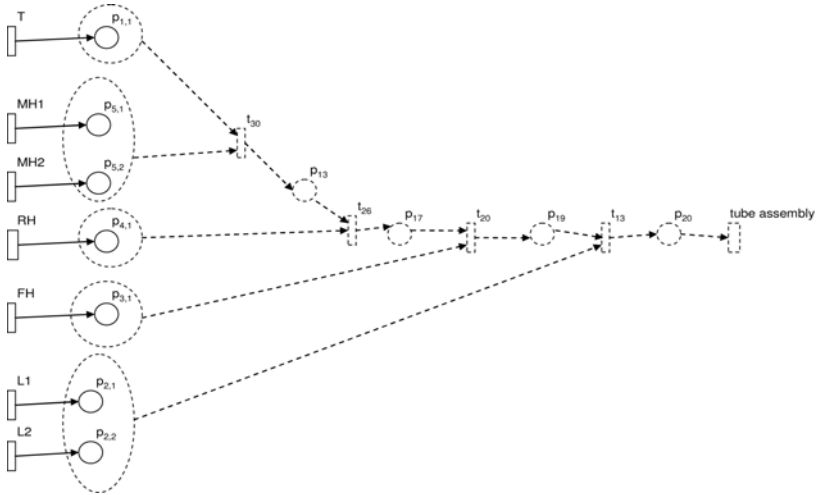


Fig. 9. Transformation corresponding to the fabrication operations

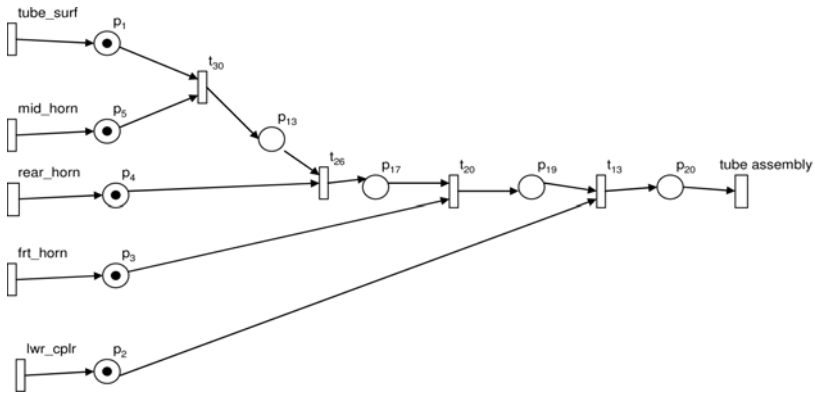


Fig. 10.  $N_i$  marking in the first step of the transformation to the product variation net

use dashed lines to represent elements of the net being replaced and solid lines for the new net. We omit the superscripts corresponding to the specific net type to avoid overcrowding the figure: the number of subscript indices indicates the specific model.

The rest of the process is guided by the execution of  $N_i$ , starting with the marking that is obtained by firing all the source transitions corresponding to fabrication operations. Figure 10 shows the  $N_i$  marking at this point of the process. At each step of the execution, we select the enabled transition in  $N_i$  (excluding transitions from  $T_f$ ), and we unfold this transition and its output place into a set of transitions for specific assembly operations in  $\mathcal{O}$  and a set of output places of subassembly

variants in  $\mathcal{H}$ . Next, we fire the transition  $N_i$  by removing the tokens from the transition input places and adding a token to the output place.

The only enabled transition at the stage shown in Fig. 10 is  $t_{30}$ , which corresponds to the assembly operation. When the transition to be unfolded represents a subassembly operation, we replace it with a set of transitions that correspond to valid assembly operations in  $\mathcal{O}$ . Assume that we replace transition  $t_r$  from  $N_i$  and that  $p_k$  and  $p_n$  are the input places of  $t_r$ , and  $p_f$  is an output place. Next,  $P_k^1$  and  $P_n^1$  are the sets of the new places that replaced  $p_k$  and  $p_n$ . Then, the set of transitions replacing  $t_r$  corresponds to all valid combinations of input places where  $p_{u,y}^1 \in P_k^1$  and  $p_{r,y}^1 \in P_n^1$ , such that  $\mathcal{H}_{p_{u,y}} \cup \mathcal{H}_{p_{r,y}} \in \mathcal{H}$ . Figure 11 shows the result of replacing  $t_{30}$  with two corresponding transitions,  $t_{30,1}^1$  and  $t_{30,2}^1$ , whose output places correspond to the specific subassemblies. Figure 12

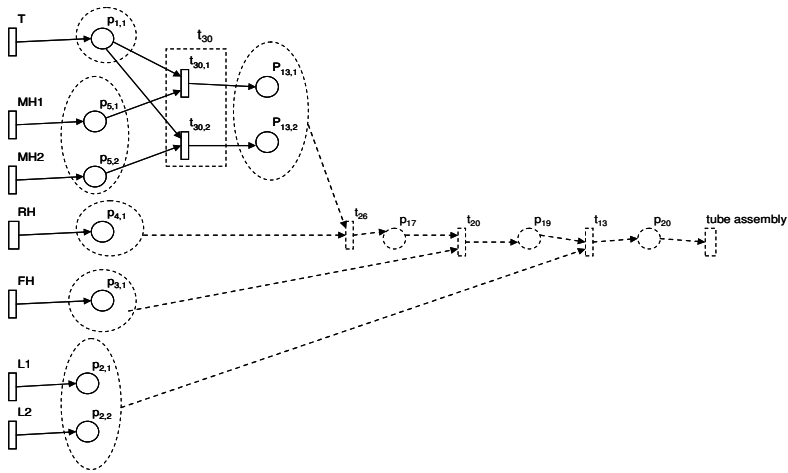


Fig. 11. Example transformation of the transition  $t_{30}$  corresponding to the assembly operation

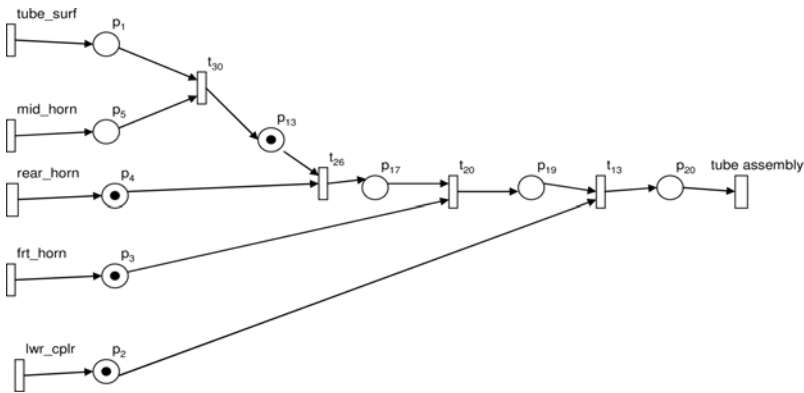


Fig. 12.  $N_i$  marking at the second step in transforming to the product variation net

presents the status of  $N_i$  at this stage. The next enabled transition is  $t_{26}$ , and so on. When the transformation process reaches the transition  $t_c$  that represents the consumption operation, we replace this transition with a set of transitions representing the consumption of the specific product variants. Figure 13 shows the final product variant net.

The size of the resulting product variation net depends on the product complexity as it defined by the EBOM and the selected t-invariant. Although the number of Petri net elements associated with component fabrications will be the same for all product variation nets, the number of places and transitions associated with intermediate subassemblies, in general, will be different for different t-invariants. For example, the product variation net in Figure 13 is based on the t-invariant that includes source transitions, sink transition, and  $t_{30}, t_{26}, t_{36}, t_{20}, t_{13}$  transitions, has a total of sixteen places (7 associated with fabrication components, 3 associated with 3 different finished products, and 6 associated with intermediate subassemblies). However, the product variation net based on the t-invariant that includes the source transitions, sink transition, and  $t_{22}, t_{16}, t_{36}, t_{13}$  has a total of fourteen places because there are only 4 places associated with intermediate subassemblies. The supply chain design implication is that the assembly process for the latter case facilitates postponement of product differentiation.

### 3.3 Supply Chain Configuration Net

This section describes the transformation from the product variation net to the supply chain configuration net. We define the supply chain configuration net graph as a 3-tuple  $N^2 = (P^2, T^2, W^2)$ , where

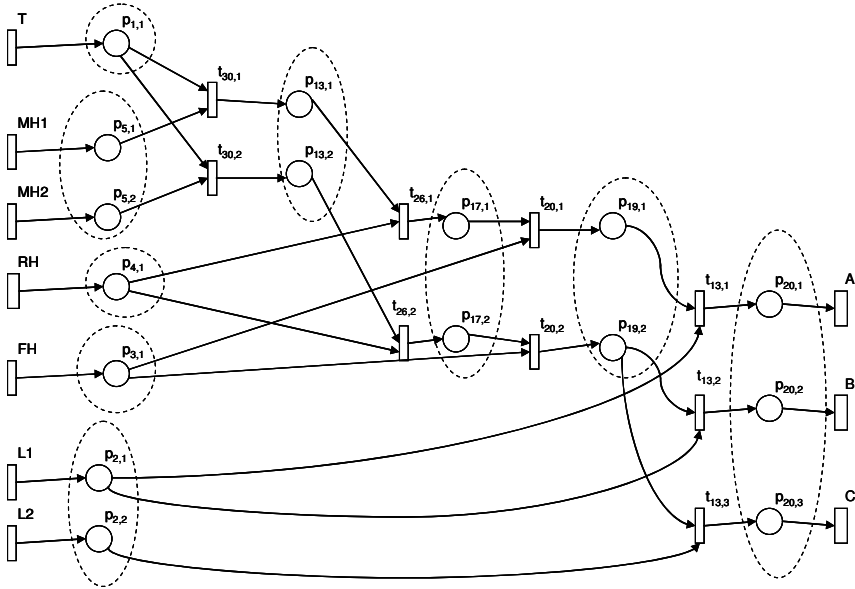
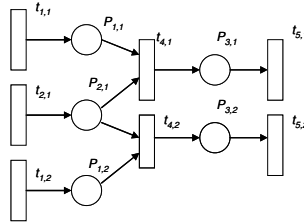


Fig. 13. Final product variant net derived from  $N_i$

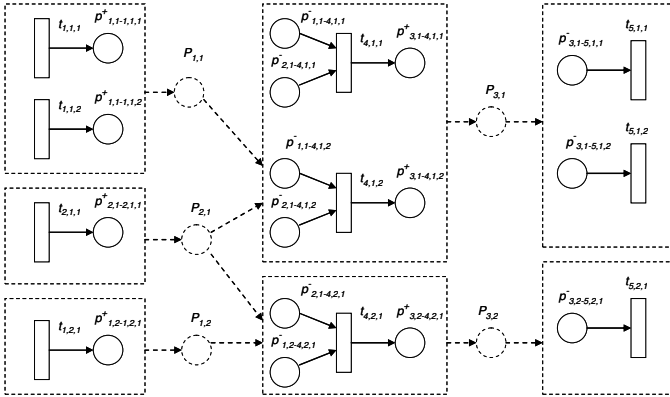
- $P^2$  is a finite set of places corresponding to a specific product component, subassembly, or final assembly at the given supply chain site.
- $T^2 = T^M \cup T^T$ , where  $T^M$  is a finite set that is composed of transitions associated with manufacturing (fabrication, assembly, and consumption) operations at a given supply chain site, and  $T^T$  is a set of transitions associated with transportation of the given component or subassembly between supply chain sites.
- $W^2 : (P^2 \times T^2) \cup (T^2 \times P^2) \rightarrow \{0, 1\}$  defines the set of directed arcs in the net.

We implement the transformation from the product variation net to the supply chain configuration net by sequentially applying the two rules below. We illustrate the application of these rules using Fig. 14, which shows an example product variation net for two product variants.



**Fig. 14.** Example of a product variation net  $N^1$  for two product variants

- Rule 1: Replace each transition  $t_{i,j}^1$  in the product variation net  $N^1$  that corresponds to the given fabrication, assembly or consumption operation with the subnets that correspond to the given operation at a specific fabrication supplier  $f \in \mathcal{F}$ , assembly supplier  $s \in \mathcal{S}$ , or assembly plant  $a \in \mathcal{A}$ : the index  $i$  refers to the transitions in the assembly, fabrication, and consumption operations, and  $j$  refers to the specific variant of the component or subassembly. Each of these subnets includes the new transition  $t_{i,j,h}^2$  together with input and output places as defined for transition  $t_{i,j}^1$  in  $N^1$ : the index  $h$  refers to the specific manufacturing facility (i.e., fabrication or assembly supplier). We use  $p_{k,l-i,j,h}^{2\omega}$  to denote the output place ( $\omega = +$ ) or input place ( $\omega = -$ ) for the transition  $(i, j, h)$ . The transition  $(i, j)$  in  $N^1$  corresponds to a specific manufacturing operation, while  $(i, j, h)$  is the transition in  $N^2$  that corresponds to the assignment of this operation to a specific supplier or assembly plant: the indices  $(k, l)$  refer to the corresponding place from  $N^1$ . Figure 15 illustrates the result of applying Rule 1 to the product variation net in Fig. 14.



**Fig. 15.** Demonstration of the transformation step of the product variation net by replacing transitions corresponding to manufacturing operations with transitions corresponding to the assignment of these operations to specific facilities

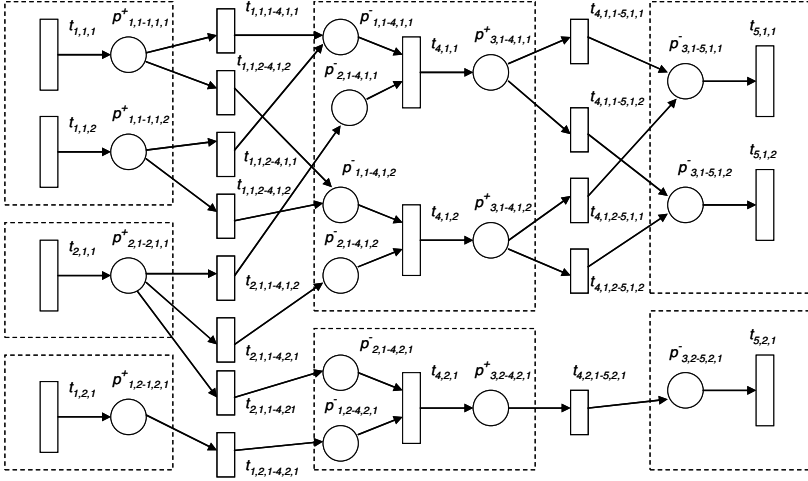
- Rule 2: Iterate through the places in the set  $P^1$  of  $N^1$ : for each  $p_{k,l}$  in  $N^1$ , create transitions linking each  $p_{k,l-i,j,h}^{2+}$  to  $p_{k,l-m,n,g}^{2-}$  in  $N^2$  for all  $(i, j, h)$  and  $(m, n, g)$ . Here,  $(i, j, h)$  refers to suppliers capable of performing assembly or fabrication operations producing components or subassemblies corresponding to transition  $t_{i,j}^1$  in  $N^1$ , and  $(m, n, g)$  refers to the supply chain sites (assembly plants or assembly suppliers) capable of performing operations represented by  $t_{m,n}^1$  in  $N^1$ . Figure 16 shows the final supply chain configuration net.

### 3.4 Integer Program Supply Chain Configuration Formulation

In the previous section, we discussed how to generate the supply chain configuration net for each individual assembly process. Recall that  $N^2 = (P^2, T^2, W^2)$ . We define  $P_d^2 \subseteq P^2$  as the set of all input places for the transitions that correspond to the consumption operations at a given assembly plant. Next, we define the supply chain configuration net markings as follows:

- $M : P^2 \rightarrow \mathbb{N}$  is the supply chain configuration net markings that represents different parts at different supply chain sites.
- $M_0 : P^2 \rightarrow \{0\}$  is the initial marking, which consists of all zeros.
- $M_d : (P_d^2 \rightarrow \mathcal{D}) \cup (P^2 \setminus P_d^2) \rightarrow \{0\}$  is the target marking that associates the demand of each assembly plant for the given product variant to the input places of the transitions associated with assembly consumption, while the rest of the places have zero marking.

In terms of Petri nets, the supply chain configuration problem consists of finding the number of firings for each transition in our net that will change the state of



**Fig. 16.** Final supply chain configuration net for the example in Figure 14

the net from an  $M_0$  marking to an  $M_d$  marking. Using a Petri net linear algebra representation, this problem can be formulated as

$$\mathcal{W}^T \tau = \mu_d - \mu_0, \quad (1)$$

where  $\mathcal{W}$  is the incidence matrix of  $N^2$ ,  $\tau$  is a firing vector, and  $\mu_d$  and  $\mu_0$  are the initial and target marking vectors, respectively, of the supply chain configuration net. In general, the solution for  $\tau$  is only a necessary condition of reachability from  $\mu_0$  to  $\mu_d$ . However, our supply chain configuration net,  $N^2$ , is acyclic because the net model represents the progressive stages of the product assembly. In the case of an acyclic net, the solution to (1) is both necessary and sufficient [15].

We can easily augment this system of linear equations with the following capacity constraints. The capacity load for a supplier can be represented by the sum of all or a subset of the transition firings of the transitions corresponding to the operations at the given supplier multiplied by the load factor associated with this operation. Let  $L$  be a  $0/1$  ( $(\sigma + \alpha) \times |T^2|$ ) matrix representing the supplier transition relationship, where

$$L_{o,m} = \begin{cases} l_{o,m} & \text{if the transition } \tau_i \text{ represents the operation } o \text{ at supplier } m \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

for every  $\tau_i \in T^2$ ,  $o \in \mathcal{O}$ , and  $m \in \mathcal{S} \cup \mathcal{F}$ . As a result, our supplier capacity constraint is as follows:

$$L\tau \leq q_s \quad \forall s \in \mathcal{S} . \quad (3)$$

Finally, we can assign the cost for each transition with the corresponding component fabrication cost ( $c_o^F$ ), assembly operation ( $c_{os}^A$ ), and transportation costs ( $c_{zs1,s2}^T$ ) to the cost vector,  $c$ . Note that all costs are piece costs for fabrication, subassembly, or transportation. They are constant inputs, resulting in a

linear objective function. Costs for assembly or fabrication are typically provided by supplier quotes or can be estimated from historical costs of similar parts and supplier profile. Transportation costs are derived based on distance and transportation rates for a given pair of locations divided by the part quantity per truck, which can be estimated using packaging tools (e.g. Design for Density [16]).

The resulting integer linear program is as follows:

$$\begin{aligned} \min_{\tau} \quad & c^T \tau & (4) \\ \text{subject to} \end{aligned}$$

$$\mathcal{W}^T \tau = \mu_d - \mu_0 \quad (5)$$

$$L\tau \leq q_s \quad \forall s \in \mathcal{S} \quad (6)$$

$$\tau \in \mathbb{N}$$

The solution to this IP provides minimum cost supply chain configuration alternatives for each process plan. Based on this initial minimum cost estimate, we can filter a subset of the best solutions for further decision analysis: we can use the generated Petri net models for the selected solutions to analyze other parameters that may be involved in the decision making.

## 4 Summary

This paper discusses a Petri net methodology for modeling and analysis of integrated process planning and supply chain configuration for assembly products. This development was inspired by a need in the automotive industry for efficient and effective tools to analyze and design supply chains for commodity assemblies; however, this methodology can be adopted for a wide variety of products involving assembly operations.

In this paper, we discussed our modeling approach based on a Petri net techniques. We started with a Petri net model representation of the assembly process for a given product. All feasible assembly sequences can be obtained as the set of minimal t-invariants of this net model. To ensure a single assembly sequence for different product variants and paths, we iterated over all feasible assembly sequences. Then, we take each individual t-invariant based subnet through a set of stepwise transformations starting with the Petri net model representing the product variation assembly net. Alternatively, this transformation could be represented by a color Petri net: this is a common approach for concise representation of systems with multiple products. However, in our development, we used one P/T net class because it simplified our software development process, and the modeling process is carried out automatically. From the product variation net, we transform to the detailed supply chain configuration model. Using an algebraic linear representation for this Petri net, we directly formulate the IP: the IP solution provides the minimal cost.

Petri net representations provide the following substantial advantages:

1. a clear and concise representation of the assembly sequence search space;
2. analysis and generation of feasible assembly sequences through t-invariants;
3. the capability to refine the model from a generic assembly sequence to a detailed supply chain configuration through a set of stepwise transformations;
4. an algebraic representation that automatically formulates the IP.

Furthermore, Petri net modeling provides us with a powerful analytical and simulation framework for studying a multitude of additional parameters, such as inventory, supply chain risk, or responsiveness.

This methodology has been incorporated into a decision support system that analyzes supply chain commodity assemblies to assist material planning, logistics, and purchasing. This system enables optimization analysis to be applied to a broad number of commodity assemblies: only a handful of critical or complex commodities can be analyzed in detail without an automated support system. Leveraging Petri net techniques allowed us to reduce the modeling effort and improve the interactive analysis capability.

## References

1. Geunes, J., Pardalos, P.: Applied Optimizaiton 98, 265–305 (2005)
2. Rosell, J., Munoz, N., Gambin, A.: Robot Tasks sequence Planning using Petri Nets. In: Proceedings of the 5th IEEE International Symposium on Assembly Task Planning, Besancon, France, July 10-11, pp. 24–29 (2003)
3. Viswanadham, N., Ragavhan, N.: Performance analysis and design of supply chains: a Petri net approach. *J. Oper. Res. Soc.*, 1158–1169 (2000)
4. You, X., Zhang, L., Jiao, J.: Supply chain configuration modeling based on colored Petri-nets. In: Proceedings of IEEE International Conference on Management of Innovation and Technology, pp. 921–925 (2006)
5. Li, Y., Tang, D., Hu, S., Chen, Y.: Route Choice to Supply Chain Based on Petri Net. In: Proceedings of the 6th World Congress on Intelligent Control and Automation, Dalian, China, June 21-23, pp. 6836–6839 (2006)
6. Zimmermann, A., Knoke, M., Yee, S., Tew, J.: Model-Based Performance Engineering of General Motors' Vehicle Supply Chain. In: IEEE Int. Conf. on Systems, Man and Cybernetics, Montreal, October 2007, pp. 1415–1420 (2007)
7. Kanehara, T., Suzuki, T., Inaba, A., Okuma, S.: On Algebraic and Graph Structural Properties of Assembly Petri Nets. In: Proceedings of 1993 Int'l. Conf. on Intelligent Robotics and Systems, pp. 2286–2293 (1993)
8. Caselli, S., Zanichelli, F.: On Assembly Sequence Planning using Petri Nets. In: Proceeding of the IEEE International Symposium on Assembly and Task Planning, pp. 239–244 (1995)
9. Urbášek, M.: Modeling Petri Net Based Systems by Net Transformations: New Developments. *Electronic Notes in Theoretical Computer Science* 82(7), 16–33 (2003)
10. de Mello, L., Sanderson, A.: A Correct and Complete Algorithm for the Generation of Mechanical Assembly Sequences. *IEEE Transactions on Robotics and Automation* 7(2), 228–240 (1991)



11. Cao, T., Sanderson, A.: AND/OR Net Representation for Robotic Task Sequence Planning. *IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews* 28(2), 204–218 (1998)
12. IBM ILOG CPLEX (2010), <http://www-01.ibm.com/software/integration/optimization/cplex/>
13. Martinez, J., Silva, M.: A Simple and Fast Algorithm to Obtain All Invariants of a Generalized Petri Nets. In: *Application and Theory of Petri Nets, Informatik-Fachberichte*, vol. 52, pp. 301–310. Springer, Berlin (1982)
14. Gusikhin, O., Kulinitch, A.: Animated AI-Based Simulation in Production Scheduling. In: Metzgar, I., Bertok, P. (eds.) *IFIP Transactions on Knowledge Based Hybrid Systems*, pp. 165–177. North-Holland, Amsterdam (1993)
15. Murata, T.: Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE* 77(4), 541–580 (1989)
16. Filev, O., Devarajan, V., Angotti, S., Taverna, J.: System and Method of Interactively Optimizing Shipping Density For a Container. Patent application number: 20090299790 (2009)
17. Weber, M., Kindler, E.: The Petri Net Kernel. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) *Petri Net Technology for Communication-Based Systems*. LNCS, vol. 2472, pp. 109–124. Springer, Heidelberg (2003)

# The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification<sup>\*</sup>

Christine Choppy<sup>1</sup>, Anna Dedova<sup>1</sup>, Sami Evangelista<sup>1</sup>, Silien Hong<sup>2</sup>,  
Kais Klai<sup>1</sup>, and Laure Petrucci<sup>1</sup>

<sup>1</sup> LIPN — Laboratoire d'Informatique de l'Université Paris Nord  
99, av. J-B Clément, 93430 Villetaneuse, France  
{firstname.lastname}@lipn.univ-paris13.fr

<sup>2</sup> LIP6 — Laboratoire d'Informatique de Paris 6  
104 av. du Pdt Kennedy, 75016 Paris, France  
silien.hong@lip6.fr

**Abstract.** This paper presents the modelling process and first analysis results carried out within the NEOPPOD project. A protocol, NEO, has been designed in order to manage very large distributed databases such as those used for banking and e-government applications, and thus to handle sensitive data. Security of data is therefore a critical issue that must be ensured before the software can be released on the market.

Our project aims at verifying essential properties of the protocol so as to guarantee such properties are satisfied. The model was designed by reverse-engineering from the source code, and then initial verification was performed. This modelling work requires choices of adequate abstraction levels both at the modelling and verification stages. In particular, the overall system is so large that the model should be carefully built in order to make verification possible without getting too far from the actual protocol implementation. This paper focuses on the modelling and initial validation of the election process launched at the system initialisation.

## 1 Introduction

The evolution of nowadays systems is characterised by an increasing complexity and an increasingly critical role. E-government, banks, internet information systems, commerce registries, ... Computers play an essential and growing role in these sensitive sectors and must access and maintain huge databases. In these activities, where the slightest defect may lead to a disaster, safety is paramount. They are characterised not only by the huge amount of data they manipulate, but also by a mandatory high level of security and reliability. The development of such applications is a complex problem which requires to elaborate reliable and safe distributed database management software. Thus, it is advisable to use formal description techniques to clearly specify the behaviour of the considered applications. It is also recommended to have automatic or semi-automatic verification tools to validate these applications.

ZODB, the *Zope Object Database* [4], has become within a few years the most used object database. This free software, associated with the Zope application server is used

---

<sup>\*</sup> This work is supported by FEDER Île-de-France/ System@tic—libre software.

for a Central Bank, to manage the monetary system of 80 million people in 8 countries [7]. It is also used for accounting, ERP, CRM, ECM and knowledge management. It is now a major free software as PHP or MySQL is.

However, the current Zope architecture does not yet handle data as huge as those mentioned above. In order to attain such performances, the architecture had to be revisited. It led to the design of an original peer-to-peer transaction protocol named *NEO*. This protocol must also ensure both safety and reliability, which is not easy to achieve for distributed systems using traditional testing techniques.

The aim of our work is to formally design and check the safety and the reliability of the NEO protocol. Designing an appropriate specification is a first challenge. Starting from the protocol description, a reverse-engineering process allows for extracting step-by-step a corresponding symmetric Petri net model [6]. Since the original program description is very large and well structured, it is mapped to a modular and hierarchical specification. However, in order to mimic different configurations of the cluster architecture deployed, as well as the different roles of the servers involved, w.r.t. the protocol operation, the model must also be highly parametrised. Regarding the verification step, two main problems arise: which properties of the protocol should be checked and are existing model checking approaches efficient enough?

In this project, we are interested in some critical properties of (a part of) the protocol, such as data consistency, fault recovery, detection of bottlenecks. The verification is then ensured using dedicated techniques (with CPN-AMI platform [2]) taking advantage of the characteristics of distributed systems. We experimented several tools exploiting a variety of reduction paradigms like modularity, symbolic representation (i.e. decision diagrams) and parametrisation in order to tackle the well-known state explosion problem. In fact, the NEO protocol is expected to handle clusters of 100 to 10,000 nodes, most of them being dedicated to storage. Therefore, safety and reliability are critical issues and explicit state space techniques have no chance to overcome the explosion of the state space.

This paper is organised as follows. In Section 2 we describe the general functioning of the NEO protocol, and in Section 3 our modelling approach from the code. Our model of the key feature of the NEO protocol, the election protocol, is presented in Section 4 in some detail, including the modelling of a crash, and of exception handling. A preliminary analysis of the desired properties is explained in Section 5, before a conclusion and some perspectives of this work in Section 6.

## 2 The NEO Protocol

The general context of the NEO protocol was described in [5], and recalled in Section 1. This section informally describes the general functioning of the protocol.

Different kinds of nodes play dedicated roles in the protocol, as depicted by the architecture on 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 while the *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 the other nodes of the identity of the primary master (light grey arrows in Figure 1).
- the **administration node** is used for manual setup when necessary (dashed arrow in Figure 1).
- **client nodes** correspond to the machines running applications concerned with the database objects. They thus 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.

This repartition of roles raises several issues. The physical architecture is highly reconfigurable: nodes can fail and become unavailable, they can restart or new nodes can be added. In all cases a new configuration is computed.

First, the primary master is *elected* among all master nodes. The election part of the protocol is the main focus of this paper. Even though election algorithms are well-known, a complex election mechanism has been designed as part of the NEO protocol so as to handle node failures or arrival of new nodes during the election process.

The primary master node maintains the key information for the protocol to operate:

- 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. It can be *updated dynamically* when new nodes join the system or nodes crash. The total number of partitions is fixed when the system starts. The partition table, although maintained by the primary master, is duplicated on all nodes for recovery purposes after a system crash.
- the **last transaction identifier** is used after a system failure to recover a consistent database configuration.

After the election of a primary master, a *verification phase* takes place, checking that all transactions were completely processed, and thus that the database is consistent across the different storage nodes.

Finally, the system enters its *operational state*, where requests from the clients are processed.

### 3 The Modelling Approach

When the project started, only a prototype version of the protocol was available and no associated RFC (Request For Comments) existed. Therefore, the model we developed relies on the sole implementation.

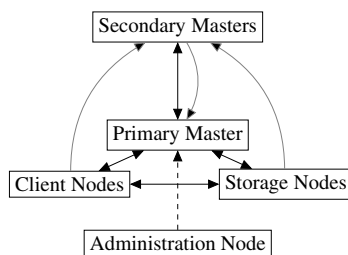


Fig. 1. The NEO-protocol topology

### 3.1 Reverse-Engineering

Since the source code of the prototype implementation was the starting point, the modelling task followed a *reverse-engineering* approach.

The source code is now available as free software, under a GPL license. It consists in 24,200 lines of Python code. The source code follows good programming practices: the code is structured in different files, each of them having a particular concern, e.g. a type of node. The code inside each file contains a few comments giving a description of what the methods are supposed to achieve, with highlights on potential subtleties. Finally, the function, object and variable names are explicit enough to ease the comprehension of the program.

The reverse-engineering approach to the Petri net model construction sticks to the program structure. This allows for:

- focusing on specific parts of the protocol;
- building a structured model, which is a key issue for verification purposes;
- an easy mapping between protocol code and Petri net model, which is necessary when verification results in undesired behaviour;
- a clear separation of concerns between the stakeholders.

Section 4 describes the model of the *election* part of the NEO protocol, including pieces of code and the associated Petri nets (see e.g. Figure 6).

Note that, during the model construction, the code evolved from a prototype implementation to a more robust, distributable version. Some rather important changes hampered our modelling process since a few design choices were revisited. To avoid such problems, we propose in Section 6 *code tagging* to track changes.

### 3.2 Abstraction Levels

Choosing adequate levels of abstraction is a key issue. First, it is necessary to *select which data structures should be modelled*, and which should not. In the NEO protocol, the nodes exchange a lot of messages. Although it is obvious that the actual data to be stored in the database should not be modelled, it is not always the case for other kinds of information contained in messages, such as the partition tables. These can be exchanged for update purposes, and such messages are important. Modelling these kinds of message easily results in state space explosion.

Second, *the analysis of the protocol is conducted in several stages*: we start by checking that the system without faults behaves as expected; then faults are introduced to check the recovery procedures after a node failure. Of course, there again, it is necessary to *identify the relevant phases of the protocol for the fault to occur*. If a failure is considered to be susceptible to happen anytime, we face state space explosion without gaining additional relevant behaviour.

The choices of abstraction levels are illustrated for the *election of primary master* part of the protocol, in Section 4. Moreover, Section 4.4 details the addition of faults handling to the model.

## 4 The Election Protocol Model

The election protocol is a key feature of the NEO system. It is triggered at the bootstrap of the cluster in order to designate among all master nodes a *primary master*. This one will play a central role in subsequent steps of the NEO protocol since it will process requests of clients to access data kept on the storage nodes. After the election, all other masters become *secondary masters*. Their role is only to be ready to replace the primary master in case it crashes. A primary master failure is immediately followed by a new election launched by the secondary master that first detects this failure.

Due to the critical aspect of this component, we developed a detailed model of the election phase in order to be able to simulate it and perform state space analysis. Since the protocol is designed to be (to some extent) fault tolerant we first focused on the ideal scenario where no fault (e.g. a master node failure, a connection loss) can occur.

Although the code of the election protocol is relatively small (around 400 lines of Python code) it turned out to be a tedious task to extract a Petri net model from it. Many high-level data structures are maintained by master nodes for synchronisation issues and a manual slicing step could only remove a small fraction of the code. Therefore, we had to make various abstractions with the primary motivation of obtaining a model amenable to state space analysis.

The Coloane environment [1] was used for modelling. It was chosen for its graphical interface, its analysis facilities using various platforms, e.g., CPN-AMI [2], and Prod [14], and its independence with respect to the underlying formalism for reasons stated 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.

Initially, a master node only knows the network addresses (IP address + port number) of its peers provided to it through a configuration file. During the negotiation step it will learn the uuids of all its peers. 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. The purpose of this first exchange is mainly related to fault tolerance as will be highlighted by the example 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 an uuid greater than its own knows it is itself the primary master.

Note that, although a master may know the identity of the primary before the end of this step (i.e., if it received an `AnswerPrim` containing the uuid of the primary master) it will still contact other masters. Indeed, the purpose of the election protocol is not

only to negotiate on the identity of a primary master but also to exchange data that are required for subsequent stages of the protocol.

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. The cluster is operational and ready to process client requests.

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.

Figure 2 is a message sequence chart describing a typical election scenario. We only depicted the message exchanges from the perspective of master M2 which will be elected as the primary master. Masters M1 and M3 naturally also have to ask for the same information. In the first exchange M2 asks M1 and M3 if they know a primary master (messages `AskPrim`). Since all masters are executing the protocol, the primary is not yet designated and they answer to M2 with a `AnswerPrim(nil)` message. M2 then requests from M1 and M3 their uuid (messages `RequestId`) which are sent back in messages `AcceptId`. Upon reception of these two messages, M2 knows it is itself the primary since all uuids requested are smaller than its uuid. It can then announce its election to M1 and M3 using an `AnnouncePrim` message. Later on, master M1 crashes. After its reboot, it asks M2 and M3 if they know a primary. They both reply that M2 is the master with uuid 897.

This example highlights the fact that entering the election phase is a local decision made by a master at its startup or if it detects the primary crash (or if it loses its connection to it). Thus some master(s) may be in election mode while others are executing the normal protocol.

The `electPrimary` method of Figure 3 implements the election algorithm<sup>1</sup>. It basically consists of four parts: the initialisation of some data structures used in the election process (ll. 2–11); the negotiation part (ll. 12–15); and the code executed by the primary master (ll. 16–21) and by secondary masters (ll. 22–24) as soon as they know their roles.

Among initialised data structures we notice an event manager `em` (l. 6) used to poll the network, a boolean `primary` specifying if the node is the primary master (l. 7) and,

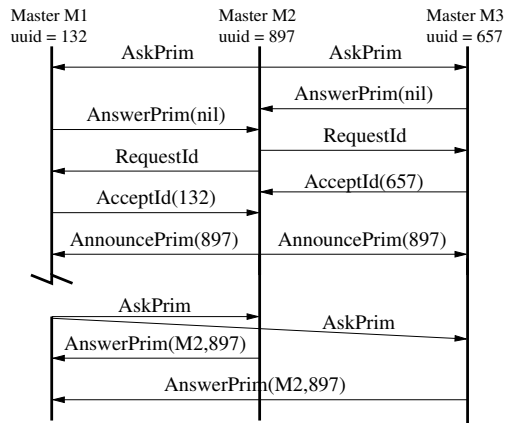


Fig. 2. Message sequence chart describing an election scenario followed by a crash and reboot of master M1

<sup>1</sup> As mentioned above, we dealt with 400 lines of Python code which we do not entirely provide here due to space constraints.

```

1 def electPrimary(self):
2     self.unconnected = set()
3     self.negotiating = set()
4     self.listening_conn.setHandler(election.
5         ServerElectionHandler(self))
6     client_handler = ClientElectionHandler(self)
7     em = self.em
8     self.primary = None
9     self.primary_master = None
10    for node in node_nm.getMasterList():
11        if node.isRunning():
12            self.unconnected.add(node.getAddress())
13            while len(self.unconnected)+len(self.negotiating)>0:
14                for addr in list(self.unconnected):
15                    ClientConnection(em, client_handler, addr=addr,
16                        connector.handler=self.connector_handler)
17                    em.poll()
18                if self.primary is None:
19                    self.primary = True
20                for conn in em.getClientList():
21                    conn.notify(AnnouncePrimary())
22                while em.getClientList():
23                    em.poll()
24            else:
25                while self.primary_master is None:
26                    em.poll()

```

**Fig. 3.** The `electPrimary` method of the Master class

otherwise, the network address of the primary (`primary_master`, l. 8). In addition, two sets identify all masters the node is not connected to and has to do so (`unconnected`, l. 2) or is negotiating with (`negotiating`, l. 3). The termination of the negotiation step (l. 12) 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 (the corresponding code of these events is not fully shown in this paper):

- initially  $\Rightarrow$  `m` puts in the `unconnected` set all masters it considers as alive (ll. 9–11 of method `electPrimary`).
- connection (attempted at ll. 13–14 of method `electPrimary`) of `m` is accepted by `n`  $\Rightarrow$  `m` moves `n` from set `unconnected` to set `negotiating`.
- `m` receives an `AcceptId` message from `n`  $\Rightarrow$  `m` discards `n` from set `negotiating`.
- `m` detects the crash of master `n`  $\Rightarrow$  `m` automatically deletes `n` from both sets.

The negotiation is done by repeatedly attempting to connect other master nodes to send them an `AskPrim` message (ll. 13–14); and handling received messages (l. 15).

Once the negotiation is finished, `self.primary` is still equal to `None` if the master did not receive any `AcceptId` message with a greater uuid and the node knows it is the primary master. It broadcasts an `AnnouncePrim` message to its peers (ll. 18–19) and keeps treating requests of other masters until all are aware of its existence (ll. 20–21). Otherwise, if the node is a secondary master, it keeps polling the network until it receives the announcement of the primary (ll. 23–24) and then disconnects itself from the primary master (not shown on the figure).

## 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 method of Figure 3 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 election process ends and the masters start a new election.

Modules are dependent and composed using two classical rules [11]: *place fusion* that merges two instances of the same homonym place; and *transition substitution* that refines a meta-transition via its replacement by a subnet modelling the details of the



transition. Such subnets always have two specific transitions start and end corresponding to the start and the end of the activity.

In all figures meta-transitions appear in red (see transitions in Figures 5(b), 7(a) — except for transition die, and 8(b)). Guards are put in small notes linked to the corresponding transition (see Figure 6). Finally, some arc labels, markings or guards are dependent 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. Lastly, places are coloured in such a way that all instances of the same place have the same colour.

A composition tool has been developed within this project that can assemble several modules through different transformations including the two mentioned above. From several modules and a composition file describing how to combine these, this tool produces a single net resulting from the composition.

An example of composition file can be seen on Figure 4. A composition file consists of a series of net definitions, the resulting net being the last one defined. Other nets are only used to ease the definition of the final one. The figure only depicts the definition of the electPrimary net that can be seen on Figure 7(a). The first step is to define the subnets that will be used to define the net (ll. 4–23). A subnet can be loaded from a file (tag fromFile) or from a previously defined net (tag fromNet). Following these subnet definitions, we have the list of operations performed to produce the net (ll. 24–38). The first operation substitutes the meta-transition poll of net electPrimary by its subnet poll loaded at l. 6 from a previously defined net. The tool is also flexible in that all operations can be conditioned by the definition (or non-definition) of some symbol(s). Here, we can see that transitions crash and primCrash modelling the crash of a master are removed from the net if the symbol faults is not defined (l. 29 and l. 33) when the tool is invoked. The last operation performed (l. 38) fuses all places sharing the same name. It is specified that initial markings of instances are composed using the sum operator. Other possibilities are available: min, max, etc.

Note that the tool is quite generic except for the fuseHomonymPlaces operation which requires to know how the fusion should operate. Hence, the tool is largely independent on the type of coloured nets.

```

1 <netComposition>
2 <!-- definition of some nets -->
3 <defineNet id="electPrimary">
4 <subNet id="electPrimary">
5 <fromFile>electPrim.model</fromFile></subNet>
6 <subNet id="poll">
7 <fromNet>poll</fromNet></subNet>
8 <subNet id="secPoll">
9 <fromNet>secpoll</fromNet></subNet>
10 <subNet id="primPoll">
11 <fromNet>primpoll</fromNet></subNet>
12 <subNet id="sendAnnPs">
13 <fromFile>sendAnnPs.model</fromFile></subNet>
14 <subNet id="sendAskPs">
15 <fromFile>sendAskPs.model</fromFile></subNet>
16 <subNet id="electionFailure" ifdef="faults">
17 <fromNet>electionFailure</fromNet></subNet>
18 <subNet id="crash" ifdef="faults">
19 <fromNet>crash</fromNet></subNet>
20 <subNet id="primCrash" ifdef="faults">
21 <fromNet>crash</fromNet></subNet>
22 <subNet id="reboot" ifdef="faults">
23 <fromNet>reboot</fromNet></subNet>
24 <substituteTrans>
25 <net>electPrimary</net>
26 <trans>poll</trans>
27 <subNet>poll</subNet>
28 </substituteTrans>
29 <deleteTrans ifndef="faults">
30 <net>electPrimary</net>
31 <trans>crash</trans>
32 </deleteTrans>
33 <deleteTrans ifndef="faults">
34 <net>electPrimary</net>
35 <trans>primCrash</trans>
36 </deleteTrans>
37 <!-- substitute other meta-transitions -->
38 <fuseHomonymPlaces method="sum">
39 </defineNet>
40 </netComposition>

```

Fig. 4. Part of the composition file for the net of Figure 7(a)

### 4.3 Detailed Specification of Some Key Elements

**General Declarations.** Figure 5(a) introduces the main colour classes we have used during the modelling of the election protocol and some places that are shared by all modules of our net. 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>.

Message types are defined by the `MSG_TYPE` class. Finally, items of class `NEG` specify the state of a negotiation between a master  $m$  and one of its peers  $p$ :

`NONE`  $\Leftrightarrow p$  has not been contacted, i.e.,  $p \in m.unconnected$ .

`CO`  $\Leftrightarrow m$  has contacted  $p$  and is waiting for its `uuid`, i.e.,  $p \in m.negotiating$ .

`DONE`  $\Leftrightarrow m$  knows the `uuid` of  $p$ , i.e.,  $p \notin m.negotiating \cup m.unconnected$ .

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. Thus:

$iam = F \wedge pm = 0 \Leftrightarrow m$  is a secondary master and does not know the primary.

$iam = F \wedge pm \neq 0 \Leftrightarrow m$  is a secondary master and thinks  $pm$  is the primary.

$iam = T \wedge pm = m \Leftrightarrow m$  is the primary master.

$iam = T \wedge pm = 0 \Leftrightarrow m$  is maybe the primary master but is still negotiating.

Tokens in place negotiation specify the content of sets `unconnected` and `negotiating` of all masters. For any pair of master  $\langle m, n \rangle$  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 treated yet there is a token  $\langle r, s, t, d \rangle$  in place `network` with  $r$  the receiver,  $s$  the sender,  $t$  (of type `MSG_TYPE`) the type of the message, and  $d$  the `uuid` encapsulated in the message (meaningful only if  $t = \text{AnsP}$ ).

Lastly, places `electionInit` (marked with  $\sum_{m \in \{1..MN\}} \langle m \rangle$ ), `electedPrimary` and `electedSecondary` model different stages of the `electPrimary` method: start of the negotiation (l. 12), start of the election in “primary mode” (l. 16) or in “secondary mode” (l. 22).

**Main Net Modelling the `electPrimary` method.** The net of Figure 5(b) is a high-level view of the `electPrimary` method. Red transitions are meta-transitions to be later substituted by the appropriate net modelling the details of the transitions. 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`). Since we do not consider faults for now, the `sendAskPs` transition is not in a loop with transition `poll`: it is useless to retry a connection that can be made at the first attempt. 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 `primPoll`). Note that we do not model here the exit of method `electPrimary` since messages

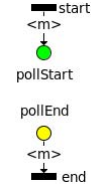
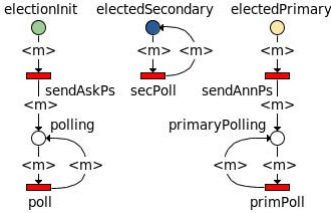
<sup>2</sup> Note that we do not distinguish in our model the `uuid` from the network address. It may however be worth modelling, in a future version, situations where a master reboots and is assigned a greater new `uuid`, as it may impact on the current election process.

```

1 parameter
2 MN = 2;
3 class
4 BOOL is [F, T];
5 M is 0 .. MN;
6 MSG_TYPE is [AskP, AnsP, R1, A1, AnnP, RP];
7 NEG is [NONE, CO, DONE];
    
```

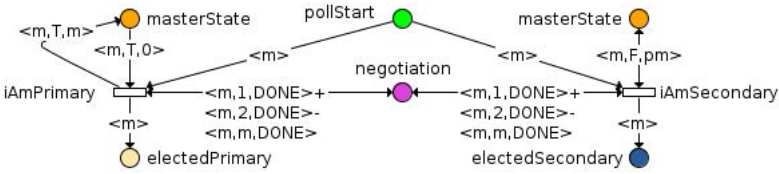
● masterState	<M,BOOL,M>	<1,T,0>+<2,T,0>
● negotiation	<M,M,NEG>	<1,2,NONE>+<2,1,NONE>
● network	<M,M,MSG_TYPE,M>	
● electionInit	<M>	<1>+<2>
● electedPrimary	<M>	
● electedSecondary	<M>	

(a) Colour classes and places shared by all modules.



(b) Main net modelling the method of Figure 3

(c) Model of the poll method.



(d) Master m has negotiated with all other masters and can decide of its role.

Fig. 5. The election protocol model

sent during this election phase may be handled during a call to the poll function made once the server master has exited the method.

A key element of the election algorithm is the poll method called by masters to handle messages received from the network. This method is called with an event manager that is attached several handlers — one for each message type — and only handles a single packet at each call by invoking the appropriate handler. It is modelled by an input transition start putting a token in place pollStart. This place is then merged with homonym places in the handler nets of Figure 6. A token <m> is present in place pollEnd if master m has finished processing a message. It can then exit the method (transition end). Specifically for the case of meta-transition poll, we also include in its subnet the nodes of Figure 5(d). These model the exit condition of the negotiation step. The negotiation is over for master m if it is not negotiating anymore with any other master: there must not be any token <m,n,neg> with neg ≠ DONE in place negotiation. Depending on the content of place masterState, the token <m> in pollStart will move to place electedPrimary or electedSecondary — both fused with their homonym places of net electPrimary (Figure 5(b)). If m has not received an AcceptId with an uuid greater than its own (see the corresponding handler in Figure 6), then a token <m,T,0> is still present in place masterState and changed to <m,T,m> since m learns it is the primary master (transition iAmPrimary). Otherwise, masterState is marked with token <m,F,pm> and m knows it is a secondary master (transition iAmSecondary).

**Message Handlers.** Nets modelling message handlers are presented in Figure 6 along with the corresponding Python code. These nets follow the same general pattern. Their transitions model the reception and handling of messages by removing one token from place `network` (the message received) and moving one token  $\langle m \rangle$  (the identity of the receiving master) from place `pollStart` to place `pollEnd`, hence specifying the message has been treated and the master can exit the `poll` function (see the net of Figure 5(c)). 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.

Handlers of types `RequestId` and `AskPrim` are rather straightforward. Therefore, we have chosen to focus on types `AnswerPrim`, `AcceptId` and `AnnouncePrim`.

For messages of type `AnswerPrim` (Figure 6(a)) we distinguish three cases.

- 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 ( $p < > 0$ ) 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.
- At 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`.

At the reception of an `AcceptId` message (Figure 6(b)), 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 5(d).

Finally, a message of type `AnnouncePrim` can be handled in two ways (Figure 6(c)) 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  $\langle m, iam, pm \rangle$  becomes  $\langle m, F, s \rangle$ .
- `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 `electPrimary` method. 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.

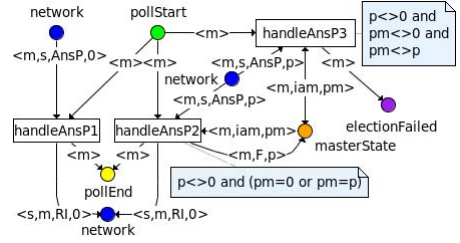
#### 4.4 Injecting Faults in the Model

Up to now, 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

```

1 def answerPrimary ( self ,
2   conn , packet , primary_uid ,
3   known_master_list ) :
4   app = self . app
5   if primary_uid is not None :
6     if app . primary is not None and \
7       app . primary_master . getUUID () != \
8         primary_uid :
9       raise ElectionFailure
10    app . primary = False
11    app . primary_master = \
12      app . nm . getByUUID ( primary_uid )
13    conn . ask ( RequestIdentification (
14      NodeTypes . MASTER ,
15      app . uid ,
16      app . server ,
17      app . name ) )

```

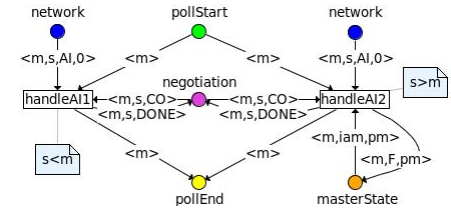


(a) Handler for message type AnswerPrim.

```

1 def acceptIdentification ( self ,
2   conn , packet , node_type ,
3   uuid , address , num_partitions ,
4   num_replicas , your_uid ) :
5   app = self . app
6   # error management
7   # ...
8   if app . uid < uuid :
9     app . primary = False
10    app . negotiating \
11      . discard ( conn . getAddress () )

```

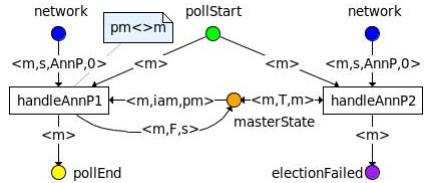


(b) Handler for message type AcceptId.

```

1 def announcePrimary (
2   self , conn , packet ) :
3   uuid = conn . getUUID ()
4   # error management
5   # ...
6   app = self . app
7   if app . primary :
8     raise ElectionFailure
9   node = app . nm . getByUUID ( uuid )
10  app . primary = False
11  app . primary_master = node

```



(c) Handler for message type AnnouncePrim.

**Fig. 6.** Message handlers and their respective models

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, starting from 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 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 5(d) 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 (almost) no consequence. 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.** The net of Figure 7(a) is the main net of Figure 5(b) modified to include the crash of a master. A fault is simply modelled by transition `crash` (resp. `primCrash`) moving token  $\langle m \rangle$  from place `electionInit` (resp. `electedPrimary`) to 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 7(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 by (or to) master  $m$ . Otherwise, if  $m$  recovers from its crash, it may handle a message received prior 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 handler nets we decided to also purge the network from messages sent by  $m$ . This is the purpose of transitions `removeRec` and `removeSen`<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 the network place is received meanwhile by another master we ensured 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 7(c). Depending on the state of  $m$  this detection has different consequences.

<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.

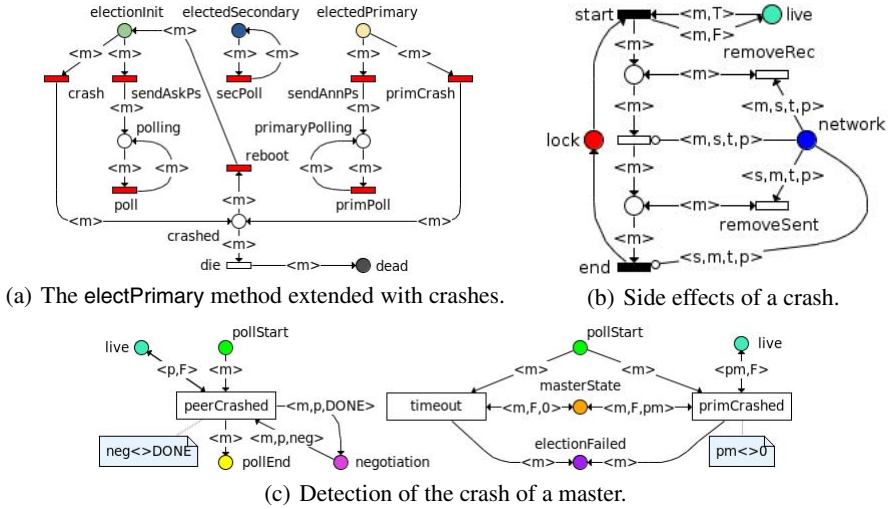


Fig. 7. Insertion of master crashes in the model

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 code perspective this consists of removing  $p$  from both  $s.unconnected$  and  $s.negotiating$ . This first situation is modelled by transition `peerCrashed` that replaces token  $\langle m,p,neg \rangle$  by  $\langle m,p,DONE \rangle$  if master  $p$  is dead, i.e.,  $\langle p,F \rangle \in live$ .

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 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.

At 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.,  $\langle pm,F \rangle \in live$  (with  $pm \neq 0$ ).

All these transitions are waiting for a token to be in place `pollStart` to become fireable. 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`.

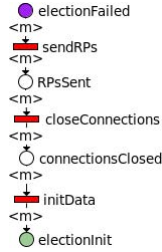
**Handling of Exception ElectionFailure.** Modelling the handling of this exception is essential if one wants to analyse the election protocol in the presence of faults. Indeed, most synchronisation issues or fault detections will be followed by this exception raise. The code for handling this exception that we had voluntarily hidden in the previous section can be seen on Figure 8(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

```

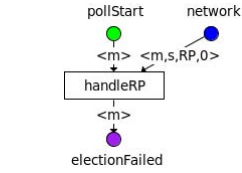
1 def electPrimary( self ):
2   ...
3   except ElectionFailure:
4     for conn in em.getClientList():
5       conn.notify(ReelectPrimary())
6       conn.abort()
7     t = time()
8     while em.getClientList():
9       and time() < t + 10:
10      try:
11        em.poll(1)
12      except ElectionFailure:
13        pass
14      for conn in em.getClientList():
15        conn.close()
16      for conn in em.getServerList():
17        conn.close()
18      # restart the negotiation

```

(a) Handler of exception ElectionFailure in the electPrimary method.



(b) Net modelling the exception handler.



(c) Handler of message type ReelectPrim.

**Fig. 8.** Modelling the handler of exception ElectionFailure

(ll. 4–6); the processing 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 negotiation by broadcasting an AskPrim message, waiting for uuids, and so on.

The corresponding net is depicted on Figure 8(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 7(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 7(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 need not to be modelled.

At last, Figure 8(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

Although we have presented a symmetric net modelling the election algorithm, we actually created two models of this protocol. The second model has exactly the same module structure but is written in the language of the Helena tool [8]. The purpose of conceiving two models is twofold. First, language Helena is richer than that of symmetric nets. Helena allows, for instance, for the use of list or set types whereas types of symmetric nets are bound to finite discrete types, e.g., enumerate types. The possibility of inserting user-defined functions in arc labels is also a useful way to easily model some problems

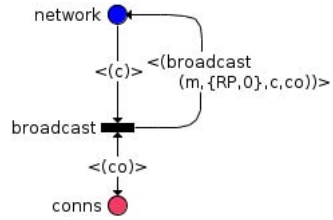


```

1 type id      : range 1 .. 3;
2 type msgType: enum (AskP, AnsP, Rf, AI, AnnP, RP);
3 type msg     : struct { msgType t; id p; };
4 type msgList: list[int] of msg with capacity 10;
5 type conns  : vector [mid,mid] of bool;
6 type chans  : vector [mid,mid] of msgList;
7 place network {dom: chans; init: <[ empty ]>> };
8 place conns  {dom: conns; init: <[ true ]>> };
9 function broadcast (mid s,msg m,chans c,conns co) -> chans {
10   for (r in mid)
11     if (s != r and co[s,r])
12       c[s,r] := c[s,r] & m;
13   return c;
14 }

```

(a) Some type and function declarations



(b) Broadcast of a ReelectPrim message

Fig. 9. Sample of the Helena model

that could, with symmetric nets, only be modelled with the use of additional transitions, irrelevant from a verification perspective. Figure 9 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 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 9(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. Table 1 lists the features and extensions to the initial model (without faults) with both languages. 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. Atomic broadcast is modelled as shown by Figure 9(b). At last, FIFO channels are implemented in the model through a list type.

We were also motivated by the fact that symbolic and explicit model checking both have their strengths and weaknesses and tools usually implement different reduction techniques to fight the state explosion. For example, although Helena can clearly not manage huge state spaces as a symbolic model checker does, it implements some form of partial order reduction which limits the exploration of redundant executions.

Lastly, this additional modelling effort was relatively small since the modelling tools we use are largely independent of the type of high-level net (this is mainly the case for Coloane, but also for our composition tool used to assemble different modules through place fusion and transition substitution). Therefore, in many cases, we only had to rewrite arc labels from one language to another, an easy task, although a bit tedious.

Table 1. Comparison of the different features provided by both models

	Node crash	Node reboot	Connection loss	Atomic broadcast	FIFO channels
Symmetric Net	✓	✓	✗	✗	✗
Helena Net	✓	✓	✓	✓	✓

## 5 Preliminary Analysis

### 5.1 Specification of Desired Properties

To ensure the system works as expected, the *desired properties* have to be specified. Hence, the engineers who implemented the protocol provided us with an *informal description* of what they consider as essential properties the protocol should satisfy.

The set of properties we were given contained 70 properties expressed in English. These had first to be refined. In particular, some properties were written in a loose manner that had to be made precise. Some terms had different meanings according to the context. Therefore the statements were re-written in order to make a consistent lot.

Analysis of the different statements allowed for characterising them:

- some of the statements were *not actual properties*. E.g. “the number of out-of-date cells per partition is greater than or equal to 0”. Here nothing can be checked since a number of cells cannot be negative.
- the model *does not handle all concerns*. E.g. the data is not modelled and thus it is impossible to prove “data consistency across partitions at all times”.
- some statements were *not accurate*. E.g. “there is a positive number of partitions”. Actually, this number *is a constant* fixed by the system administrator at the bootstrap and we can easily check that it is never changed.
- part of the properties *concern a particular kind of node*. E.g. “there is no more than 1 primary master node”. The master nodes participate in the election of one of them. This election phase must result in selecting a single primary master node. Note that the other types of nodes do not participate in the election, and therefore the verification can be achieved by concentrating on the master nodes model only, as described in Section 4.
- some tricky properties are *expressed on a particular kind of node, but actually also concern the other nodes*. E.g. “there is at least one master node for the system to operate”. If all master nodes are down, the system should stop, and thus the other kinds of nodes should not operate anymore. Hence this is a global property.
- some properties *concern a particular stage of the protocol*. For their verification, it is only necessary to focus on this particular part of the protocol operation.

Part of the properties are also only relevant at a specific abstraction level. For instance, we made a distinction between the normal operating mode and the faulty one (see Section 4.4). E.g. “after a primary master failure an election takes place”. In the normal operating mode, there is a single election at the system bootstrap. When considering primary master faults, an additional election phase occurs.

### 5.2 State Space Analysis of the Election Protocol

State space analysis has been conducted on the election model described in Section 4. 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 [2], this net was then unfolded in a low-level one using optimised techniques [9] and finally reduced [10] to produce a smaller net (but equivalent with respect to 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. 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 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.

Both R0, R1 and R2 can be expressed as a safety property 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 2. 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 markings and arcs of its state space, together with the number of terminal markings, i.e., with no outgoing arcs.

For each listed configuration we have checked whether or not requirements R0–R3 are verified. Our observations are the following ones:

- In the faultless configurations ((2,no,0) and (3,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 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.

**Table 2.** State space analysis of some configurations

Configuration			Markings	Arcs	Terminal markings	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	×	×	×	

**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.

## 6 Conclusion and Perspectives

In this paper we presented our work to model the NEO protocol and to verify some critical properties. The modelling is achieved by *reverse-engineering* from the code, and required to devise appropriate choices to work on relevant and useful *levels of abstraction at different steps*. Large applications require a modular modelling with some *composition mechanisms*, and we use the place fusion and transition substitution mechanisms, while a composition tool was developed to put this into practice. We also worked on non-nominal situations by *injecting faults* and on the way *exceptions* are handled, such as the crash of a primary master and checking their adequate detection. While the modelling is achieved using symmetric nets, an alternative modelling in language Helena allows for some additional checks. The work done on the properties started with *analysing the properties provided in natural language* so as to produce relevant and adequate properties that then could be checked.

While this work was going on, the code was changed by the developers (who did not realise that it could impact our work). This raised the *traceability* issue of our work. We

plan to address it by “tagging” the code with references to the corresponding parts of our model, thus maintaining a *clear correspondence between code and model*.

Further analysis has still to be made on the election process. Developers are gradually integrating various optimisations in the code. Some are really straightforward and should not impact the properties of the protocol while a few are rather tricky and have to be integrated in our model for verification purposes.

Although we only used CPN-AMI and Helena to analyse the election protocol (as detailed in Section 5.2), we plan to use other analysis techniques. This can be done through the facilities provided by Coloane which can interface with several tools: GreatSPN [3] which can build the symbolic reachability graph [6]; Prod [14] which implements the stubborn set technique [13] to avoid the exploration of the full state space; and Mod-SOG, a modular and symbolic model checker based on observation graphs [12].

## References

1. The Coloane tool Homepage, <https://coloane.lip6.fr/>
2. The CPN-AMI Homepage, <http://move.lip6.fr/software/CPNAMI/>
3. The GreatSPN tool Homepage, <http://www.di.unito.it/~greatspn/index.html>
4. The ZODB Homepage, <http://wiki.zope.org/ZODB/FrontPage>
5. 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)
6. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: A Symbolic Reachability Graph for Coloured Petri Nets. *Theoretical Computer Science* 176(1-2), 39–65 (1997)
7. ERP5. Central Bank Implements Open Source ERP5 in Eight Countries after Proprietary System Failed, <http://www.erp5.com/news-central.bank>
8. 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)
9. 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)
10. Haddad, S., Pradat-Peyre, J.-F.: New Efficient Petri Nets Reductions for Parallel Programs Verification. *Parallel Processing Letters* 1, 16 (2006)
11. 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)
12. Klai, K., Petrucci, L.: Modular Construction of the Symbolic Observation Graph. In: ACSD 2008, pp. 88–97. IEEE, Los Alamitos (2008)
13. Valmari, A.: A Stubborn Attack on State Explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
14. 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)

# Factorization Properties of Symbolic Unfoldings of Colored Petri Nets

Thomas Chatain<sup>1</sup> and Eric Fabre<sup>2</sup>

<sup>1</sup> LSV, ENS Cachan, CNRS, France  
thomas.chatain@lsv.ens-cachan.fr

<sup>2</sup> INRIA Rennes - Bretagne Atlantique, France  
eric.fabre@irisa.fr

**Abstract.** The unfolding technique is an efficient tool to explore the runs of a Petri net in a true concurrency semantics, i.e. without constructing all the interleavings of concurrent actions. But even small real systems are never modeled directly as ordinary Petri nets: they use many high-level features that were designed as extensions of Petri nets. We focus here on two such features: colors and compositionality. We show that the symbolic unfolding of a product of colored Petri nets can be expressed as the product of the symbolic unfoldings of these nets. This is a necessary result in view of distributed computations based on symbolic unfoldings, as they have been developed already for standard unfoldings, to design modular verification techniques, or modular diagnosis procedures, for example. The factorization property of symbolic unfoldings is valid for several classes of colored or high-level nets. We derive it here for a class of (high-level) open nets, for which the composition is performed by connecting places rather than transitions.

## 1 Introduction

Although they offer a satisfactory representation of concurrency, Petri nets are often difficult to use to model even small real systems. Their drawback is that the state of the system is only represented by the position of the tokens in the places. Consequently, in order to distinguish between the different values that a variable of the system can take, the simplest way is often to use one place per value. Even if more subtle codings are possible, the number of necessary places and transitions becomes very large, or even infinite, which makes the system very hard to comprehend. For this reason several extensions of Petri nets have been proposed, like the well-known colored Petri nets [15].

In [17], Khomenko and Koutny, developed a notion of unfoldings for high-level Petri nets, which is based on a transformation of the high-level model into a low-level model, in order to reuse the unfolding technique that was developed for low-level models. We call this method *expanded unfolding*. Our method yields a much more compact structure, where the executions are grouped into symbolic processes that reflect the generic aspect of the model. Symbolic unfoldings were already studied in [11] and [9]. Here we focus on their factorization properties: we

show how the symbolic unfolding of a large system described as a composition of several components, can be computed from the symbolic unfoldings of the components. Some factorization properties are obtained in [10], but only for high-level processes, not for branching processes or symbolic unfoldings.

We choose a framework where nets are composed via shared places, called interface places or open places when seen from one component. This way of composing nets is very popular because it matches nicely the graphical nature of Petri nets. And it has indeed been introduced in different formalisms. For example in net algebras, where nets can be composed by fusion of places [6,19,5,14]. Petri nets with interface of [21,22] use a similar construction. The composition operator of [23] is motivated by the popularity of this kind of compositions and remarks that net process are built by assembling tiles via fusion of places. Closer to our work, [18] defines a partial order semantics for Petri net components that communicate via interface places with an environment. A categorical formalization of open nets was proposed in [2,3], where open net processes are defined, but no unfolding. We discuss later the differences between this work and ours.

The paper is organized as follows. Next section introduces the net family that is used in this paper: a variant of colored Petri nets, enriched with interface places that are used to compose them. We call such nets colored puzzle nets. We introduce an adequate category setting for them and study their compositionality properties. Our morphisms differ clearly from those of [2,3], which is crucial to prepare our main result on unfoldings. Section 3 reviews and adapts a standard expansion procedure for colored nets, that separates colors in order to transform a colored net into an equivalent uncolored one. Section 4 contains the main contribution of the paper. It examines the notion of symbolic unfolding for a colored puzzle net, and studies its relation both to composition and to expansion. It is stated there that the symbolic unfolding of a product puzzle net is the product of the unfoldings of its components.

The detailed proofs can be found in [8].

## 2 Colored Puzzle (Petri) Nets and Their Composition

Colored Petri nets were defined by Jensen in [15], as one possible formalism to enhance the flexibility of Petri nets and facilitate the modeling of real systems. In these nets, each token carries some information, traditionally called the *color* of the token. Of course, transitions can test the color of the tokens they consume, and the color of the created tokens may depend on the color of the consumed tokens. These constraints on the values of the tokens are called *guards*. In this paper we are not interested in giving a precise syntax to the guards, therefore we simply describe them as sets of possible *firing modes*, each firing mode assigning a color to each input and output place of the transition. Of course our results about factorization of unfoldings remain valid in presence of syntactical restrictions for the guards.

This section introduces colored puzzle nets, or puzzle nets for short, a variant of colored nets where extra interface places are introduced. These interface places

are used for connectivity purposes. They model the communication between a component and other components or its environment, as it is the case in reactive systems. As long as the environment is not modeled, it may consume or create tokens at any time in interface places. Then it is meaningless to remember the marking in these places or to test the presence of a token in an interface place when a transition needs it to fire. For this reason interface places have a special status: they are neutral in most operations, like the computation of trajectories, the computation of unfoldings, etc. They only become active when they are connected to another component, in which case they change status and start behaving as ordinary places, and thus impose new constraints on the behavior of the component.

Labels are used for the composition, like in [13] or [1].

## 2.1 Colored Puzzle Nets

A (possibly infinite) set  $V$  of *colors* is given once for all, and is used in all the nets of the paper.

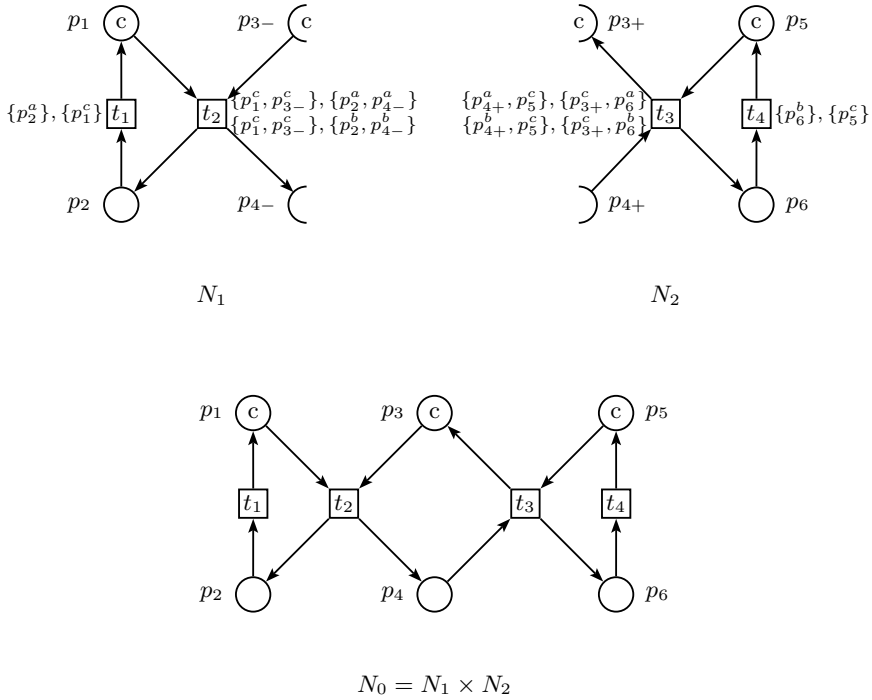
**Definition 1 (colored puzzle (Petri) net).** A colored puzzle net is a tuple  $N \stackrel{\text{def}}{=} (P, P^+, P^-, T, pre, post, \Lambda, \lambda, \gamma, M^0)$  where

- $P, P^+$  and  $P^-$  are disjoint (possibly infinite) sets of internal places, positive places and negative places respectively (think of magnet polarities); We denote  $P^\pm \stackrel{\text{def}}{=} P^+ \cup P^-$  the interface places; The polarities are needed when nets are composed by product (see Section 2.3).
- $T$  is a (possibly infinite) set of transitions;
- $pre, post : T \rightarrow 2^{P \cup P^\pm}$  map each transition  $t \in T$  to a preset often denoted  $\bullet t \stackrel{\text{def}}{=} pre(t)$  and a postset often denoted  $t^\bullet \stackrel{\text{def}}{=} post(t)$  respectively;
- $\gamma$  maps each transition  $t \in T$  to a guard  $\gamma(t)$ , that is a set of pairs  $(\alpha, \beta) \in (\bullet t \rightarrow V) \times (t^\bullet \rightarrow V)$  called firing modes ;
- the initial marking  $M^0$  is a multiset of pairs  $(p, v) \in (P \cup P^\pm) \times V$ . We sometimes write  $M^0(p)$  for the multiset of colors in place  $p$ .
- $\Lambda$  is a label set;
- $\lambda : P \cup P^\pm \rightarrow \Lambda$  assigns a label to each place, such that only internal places can share the same label, i.e. for all  $p_1, p_2 \in P \cup P^\pm$ , if  $p_1 \neq p_2$  and  $\lambda(p_1) = \lambda(p_2)$ , then  $p_1$  and  $p_2$  are internal places;

The tokens in the interface places  $P^\pm$  are not considered in the semantics of a single component, since they may be created or consumed freely by the environment. In particular, when a token of an interface place is needed to fire a transition, one must always consider that the token may have been created by the environment. Thus, a marking  $M$  for the puzzle net  $N$  is a multiset  $M : P \times V \rightarrow \mathbb{N}$ .<sup>4</sup> Transition  $t$  is firable from  $M$  with firing mode  $(\alpha, \beta) \in \gamma(t)$

<sup>4</sup> Remark that in the definition of the net, the initial marking is defined also on the interface places. The reason for this is that when we compose two nets, they have to agree on the initial marking of their shared places, that become internal places of the product.





**Fig. 1.** Two components (on top) and their composition via the product (bottom). The firing modes of the transitions are omitted on the picture of  $N_0$ . They are the same as those of the corresponding transitions of  $N_1$  and  $N_2$ .

iff  $\forall p \in \bullet t \cap P, M(p, \alpha(p)) \geq 1$ . This firing produces the marking

$$M' \stackrel{\text{def}}{=} M - \{(p, \alpha(p)) \mid p \in \bullet t \cap P\} + \{(p, \beta(p)) \mid p \in t^\bullet \cap P\}.$$

We denote  $M[(t, \alpha, \beta)]M'$ . In other words,  $\alpha$  represents the colored tokens that are consumed, and  $\beta$  represents those that are produced.

In the standard sequential semantics, a run of  $N$  is a sequence of transition firings  $M_0 [(t_1, \alpha_1, \beta_1)] M_1 \dots M_{n-1} [(t_n, \alpha_n, \beta_n)] M_n$ . Naturally, we will move to a true concurrency semantics for runs, in the sections devoted to unfoldings.

Figure 1 illustrates puzzle nets. Consider  $N_1$ : it has two internal places  $p_1$  and  $p_2$ , and two interface places  $p_3, p_4$  that are both of negative sign (mentioned in subscript). The initial marking has placed a token of color  $c$  in  $p_1$  and  $p_3$ , while  $p_2$  and  $p_4$  are empty. The transition modes are represented close to each transition.  $t_1$  has a single firing mode: it consumes a token of color  $a$  in  $p_2$ , and produces a token of color  $c$  in  $p_1$ . By contrast,  $t_2$  has two firing modes. Both consume a token of color  $c$  in  $p_1$  and in  $p_3$ . But the first mode places a token of color  $a$  in  $p_2$  and  $p_4$ , while the second mode places a token of color  $b$  in these places.

Consider now  $N_0$ , that also obeys the above constraints. Only two maximal executions are possible.  $t_2$  fires first and produces two tokens of the same color ( $a$  or  $b$ ), one in  $p_2$  and one in  $p_4$ . Then, if these tokens are of color  $a$ ,  $t_1$  and  $t_3$  can fire concurrently and the net stops; otherwise,  $t_3$  and  $t_4$  fire in sequence. Notice that because of the guards, no execution contains both  $t_1$  and  $t_4$ . Without colors these two transitions could have fired concurrently. This phenomenon will be formalized later as the notion of *color conflict*.

## 2.2 Morphisms

Before moving to the definition of composition for colored puzzle nets, we need the extra notion of morphism between two colored puzzle nets. Morphisms are relations between nets that ensure the preservation of the behaviors.

**Definition 2 (morphism).** *Let  $N_1$  and  $N_2$  be two nets such that  $\Lambda_2 \subseteq \Lambda_1$  (we add subscript  $i$  to elements of  $N_i$ ). A morphism  $\phi$  from  $N_1$  to  $N_2$  is a pair  $(\phi^T, \phi^P)$  of partial functions (the symbol  $*$  is used when the function is undefined)*

$$\begin{aligned} \phi^T &: T_1 \longrightarrow T_2 \cup \{*\} \\ \phi^P &: \begin{cases} P_1 & \longrightarrow P_2 \cup P_2^\pm \cup \{*\} \\ P_1^+ & \longrightarrow P_2^+ \cup \{*\} \\ P_1^- & \longrightarrow P_2^- \cup \{*\} \end{cases} \end{aligned}$$

such that

- $\forall p_1 \in P_1 \cup P_1^\pm \quad \begin{cases} \phi^P(p_1) = * & \text{iff } \lambda_1(p_1) \notin \Lambda_2 \\ \lambda_1(p_1) \in \Lambda_2 \implies \lambda_2(\phi^P(p_1)) = \lambda_1(p_1) \end{cases}$
- $\forall t_1 \in T_1 \quad \phi^T(t_1) = * \implies \phi^P(\bullet t_1 \cup t_1 \bullet) \subseteq P_2^\pm \cup \{*\};$
- for all  $t_1 \in T_1$  such that  $\phi^T(t_1) = t_2 \neq *$ ,
  - $* \notin \phi^P(\bullet t_1 \cup t_1 \bullet)$
  - the restriction of  $\phi^P$  to  $\bullet t_1$  is a bijection from  $\bullet t_1$  to  $\bullet t_2$
  - the restriction of  $\phi^P$  to  $t_1 \bullet$  is a bijection from  $t_1 \bullet$  to  $t_2 \bullet$
  - $\forall (\alpha_1, \beta_1) \in \gamma_1(t_1), \exists (\alpha_2, \beta_2) \in \gamma_2(t_2) : \forall p_1 \in \bullet t_1, \alpha_2(\phi^P(p_1)) = \alpha_1(p_1),$   
and  $\forall p_1 \in t_1 \bullet, \beta_2(\phi^P(p_1)) = \beta_1(p_1),$
- $\forall p_2 \in P_2 \cup P_2^\pm, \quad M_2^0(p_2) = \sum_{p_1: \phi^P(p_1)=p_2} M_1^0(p_1).$

In words,  $\phi^P$  is defined exactly on places that carry a label of the image net, and preserves these place labels.  $\phi^P$  also preserves the polarity of an interface place, and may assign a polarity to an internal place. This change of status *must* occur for places connected to a transition  $t_1$  that is removed by  $\phi^T$ , unless if such places vanish through  $\phi^P$ . When a transition  $t_1$  is preserved by  $\phi^T$ , all its connected places are preserved as well, and firing modes of this transition  $t_1$  are mapped into the modes of its image  $t_2$ . Notice in particular that  $N_1$  and  $N_2$  may be identical up to their firing modes, and the identity mapping is then a morphism as soon as  $\gamma_1 \subseteq \gamma_2$ .

From this definition, it is clear that a run of  $N_1$  is naturally mapped by  $\phi$  into a run of  $N_2$ .

The composition of two morphisms  $\phi_1 \stackrel{\text{def}}{=} (\phi_1^T, \phi_1^P)$  from  $N_1$  to  $N_2$  and  $\phi_2 \stackrel{\text{def}}{=} (\phi_2^T, \phi_2^P)$  from  $N_2$  to  $N_3$  is  $\phi_2 \circ \phi_1 \stackrel{\text{def}}{=} (\phi_2^T \circ \phi_1^T, \phi_2^P \circ \phi_1^P)$ . The identity morphism for  $N$  is  $1_N \stackrel{\text{def}}{=} (1_T, 1_{P \cup P^\pm})$ .

**Theorem 1.** *The family of colored puzzle nets equipped with the above notion of morphism forms a category.*

*Proof sketch.* Associativity and identity are straightforward. The proof of the composition can be found in [8].  $\square$

To simplify the notations we often write  $\phi$  instead of  $\phi^T$  or  $\phi^P$ . We also denote by  $N_1 \sim N_2$  the fact that  $N_1$  and  $N_2$  are isomorphic, i.e. the existence of two morphisms  $\phi_{12}$  from  $N_1$  to  $N_2$  and  $\phi_{21}$  from  $N_2$  to  $N_1$  such that  $\phi_{21} \circ \phi_{12} = 1_{N_1}$  and  $\phi_{12} \circ \phi_{21} = 1_{N_2}$ .

**Comparison with the category of open nets [23].** Forgetting high-level features like colors and guards, our puzzle nets are close to the open nets proposed in [23]. But the morphisms between them are quite different. Apart from technical aspects (we use partial functions, rather than total ones), a significant difference is that they preserve runs in the opposite direction: in our category, a morphism from  $N_1$  to  $N_2$  maps every run of  $N_1$  to a run of  $N_2$  (i.e.  $N_2$  simulates  $N_1$ ), whereas in open nets, a morphism from  $N_1$  to  $N_2$  maps every run of  $N_2$  to a run of  $N_1$  (i.e.  $N_1$  simulates  $N_2$ ).

The composition operations defined for open net and for puzzle nets are very similar in their principle: they both amount to identifying places that carry the same label. In the case of open nets, this labeling comes from the injection of a common interface net into the two components that must be assembled. In both categories, the two components that are assembled both simulate the resulting composed net. However, as morphisms and simulation relations do not have the same directions in the two settings, the composition is expressed as a pushout for open nets (a colimit), and as a product for puzzle nets (a limit).

This difference becomes crucial when coming to the construction of processes, and more generally unfoldings. In the category of open nets, there is a morphism from a process to the net  $N$ , expressing that this process simulates net  $N$ . By contrast, in the category of puzzle nets, the similar morphism from the (branching) process  $O$  to the net  $N$  expresses that net  $N$  simulates the branching process  $O$ , or more generally the unfolding of  $N$ . This choice of direction is crucial to obtain a universal property on unfoldings, which is the key to transport composition operations from nets to unfoldings. Notice that this morphism architecture reproduces the one followed in [24].

### 2.3 Product

The meaning of interface places appears in the composition of puzzle nets, that we are going to express as a categorical product. The composition mechanism is governed by the following idea, that mimics the one proposed for open nets [23]:

when two components are connected, interface places with identical labels will be merged, provided they have complementary polarities, just like magnets. Once two places are merged, the resulting pair becomes an ordinary internal place of the composed (product) net, i.e. the polarity vanishes. Notice that this notion of polarity has no relation with any idea of input or output place: interface places can communicate with the environment in any direction.

Polarities represent the part of the interface place that is owned by each component. This is why morphisms must respect polarities: an interface place can only be simulated by an interface place with the same polarity. It would be possible to deal with more than two polarities, meaning that some interface place could be shared by more than two components. An interface place could then be seen as a pie-chart, each component owning a part of the pie-chart; as long as the pie-chart would not be full, the place would keep its status of interface. When full, the place would become an internal place. This idea of multiple polarities can help one convince himself that interface places do not need to be used only as inputs or only as outputs.

**Definition 3 (product).** *Let  $N_1$  and  $N_2$  be two nets such that*

- $\forall p_1, p_2 \quad \lambda_1(p_1) = \lambda_2(p_2) \implies \begin{cases} (p_1, p_2) \in (P_1^+ \times P_2^-) \cup (P_1^- \times P_2^+) \\ M_1^0(p_1) = M_2^0(p_2) \end{cases}$
- $\forall t_i \in T_i \quad (\bullet t_i \cup t_i \bullet) \cap P_i \neq \emptyset.$

*We define their product  $N_0$ , denoted  $N_1 \times N_2$  and the associated morphisms  $\pi_1$  and  $\pi_2$  as:*

*places:*

$$\begin{aligned} P_0 &\stackrel{\text{def}}{=} (P_1 \times \{*\}) \cup (\{*\} \times P_2) \cup \{(p_1, p_2) \mid \lambda_1(p_1) = \lambda_2(p_2)\} \\ P_0^+ &\stackrel{\text{def}}{=} \{(p_1, *) \mid p_1 \in P_1^+ \wedge \lambda_1(p_1) \notin \Lambda_2\} \cup \{(*, p_2) \mid p_2 \in P_2^+ \wedge \lambda_2(p_2) \notin \Lambda_1\} \\ P_0^- &\stackrel{\text{def}}{=} \{(p_1, *) \mid p_1 \in P_1^- \wedge \lambda_1(p_1) \notin \Lambda_2\} \cup \{(*, p_2) \mid p_2 \in P_2^- \wedge \lambda_2(p_2) \notin \Lambda_1\} \\ \pi_i^P((p_1, p_2)) &\stackrel{\text{def}}{=} p_i \text{ (even when } p_i = *). \end{aligned}$$

*labels:*

$$\begin{aligned} \Lambda_0 &\stackrel{\text{def}}{=} \Lambda_1 \cup \Lambda_2 \\ \lambda_0((p_1, p_2)) &\stackrel{\text{def}}{=} \lambda_i(p_i) \text{ when } p_i \neq *. \\ \text{Notice that the restriction of } \pi_i^P \text{ to } \lambda_0^{-1}(\Lambda_i) &\text{ is a bijection to } P_i \cup P_i^\pm. \end{aligned}$$

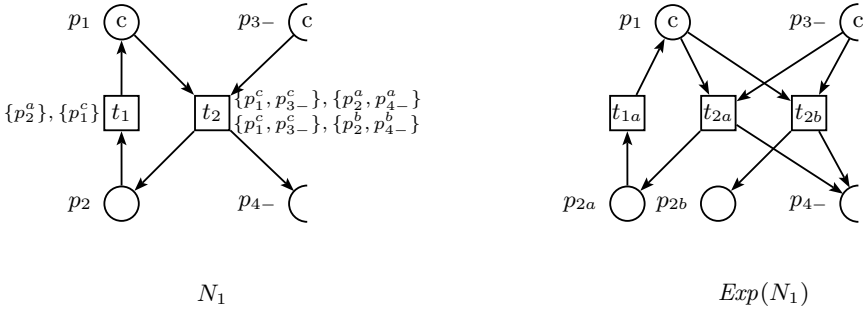
*initial marking:*

$$M_0^0((p_1, p_2)) \stackrel{\text{def}}{=} M_i^0(p_i) \text{ when } p_i \neq *$$

*transitions:*

$$\begin{aligned} T_0 &\stackrel{\text{def}}{=} (T_1 \times \{*\}) \cup (\{*\} \times T_2) \\ \pi_i^T((t_1, t_2)) &\stackrel{\text{def}}{=} t_i \text{ (even when } t_i = *) \\ \bullet(t_1, t_2) &\stackrel{\text{def}}{=} \pi_i^{-1}(\bullet t_i) \text{ and } (t_1, t_2)^\bullet \stackrel{\text{def}}{=} \pi_i^{-1}(t_i^\bullet) \text{ when } t_i \neq * \\ \gamma(t_0) &\stackrel{\text{def}}{=} \{(\alpha \circ \pi_i|_{\bullet t_0}^{-1}, \beta \circ \pi_i|_{t_0^\bullet}^{-1}) \mid (\alpha, \beta) \in \gamma(\pi_i(t_0))\} \text{ when } \pi_i(t_0) \neq * \end{aligned}$$

Observe that this composition takes the disjoint union of transitions, by contrast with several alternate notions of product for Petri nets that rather synchronize transitions. Since transitions remain private, their flow is preserved, as



**Fig. 2.** A colored puzzle net (left) and its expansion (right)

well as their firing modes, up to the reshaping of place names performed by the composition.

Figure 1 illustrates the composition by product. Places labeled  $p_3$  in  $N_1$  and  $N_2$  are merged in the product, because they have complementary polarities (the product would be undefined if they had identical polarities). The same holds for places labeled  $p_4$ . So nets  $N_1$  and  $N_2$  are assembled by these interface places, that now change their status to internal places.

**Theorem 2.** Definition 3 corresponds to the categorical product in the category of puzzle nets (Definitions 1 and 2).

*Proof sketch.* One has to check that  $N_0$  is a net: the non-trivial part is to show that only internal places share labels. Then we show easily that  $\pi_i$  is a morphism from  $N_0$  to  $N_i$ .

Finally it remains to check the universal property of the product in this category: for any  $N$  and any pair of morphisms  $\phi_i : N \rightarrow N_i$ , there exists a unique morphism  $\psi$  from  $N$  to  $N_0 = N_1 \times N_2$  that makes the diagram commutative, i.e. that satisfies  $\phi_i = \pi_i \circ \psi$ . This  $\psi$  is necessarily defined by:

$$\forall x \in P \cup P^\pm \cup T \quad \psi(x) \stackrel{\text{def}}{=} \begin{cases} * & \text{if } \phi_1(x) = \phi_2(x) = * \\ (\phi_1(x), \phi_2(x)) & \text{otherwise.} \end{cases}$$

It is straightforward to check that  $\psi$  satisfies Definition 2. □

### 3 Expansion

There exists a classical method to expand colored Petri nets into ordinary (or low-level) nets, which sometimes motivated the use of high-level nets as convenient generators of low-level models. This expansion operation is called *unfolding* by some authors [6]. In this paper, we prefer to call it *expansion*, and reserve the term *unfolding* for its standard meaning, since both operations will be simultaneously applied to colored puzzle nets.

**Definition 4 (expanded net).** *An expanded net is a colored puzzle net  $N$  such that:*

- $\forall t \in T, \quad |\gamma(t)| = 1$  (the unique element of  $\gamma(t)$  is denoted  $(\alpha_t, \beta_t)$ )
- $\forall p \in P \quad |\{\alpha_t(p) \mid p \in \bullet t\} \cup \{\beta_t(p) \mid p \in t^\bullet\}| = 1$  (the unique color in this set is denoted  $col(p)$ )

In other words, transitions have a single firing mode, and places can carry tokens of a single color of  $V$ , which coincides with the mode of all connected transitions. Notice that the second condition doesn't apply to interface places: they are not expanded into their different colors, which corresponds to the idea that a puzzle net should not restrict the set of colors in a place that will eventually be shared with another component. Anticipating a little, interface places will not be duplicated either by the unfolding procedure.

**Definition 5 (expansion).** *Given a colored puzzle net  $N$ , we define its expansion  $Exp(N) \stackrel{\text{def}}{=} (P', P^+, P^-, T', pre', post', \Lambda, \lambda', \gamma', M^{0'})$  and the associated compression morphism  $\chi_N : Exp(N) \rightarrow N$  as:*

*places:*

$P^+$  and  $P^-$  are the sets of interface places of  $N$

$$P' \stackrel{\text{def}}{=} \{(p, v) \in P \times V \mid \exists t \in T, (\alpha, \beta) \in \gamma(t) \\ (p \in \bullet t \wedge \alpha(p) = v) \vee (p \in t^\bullet \wedge \beta(p) = v)\}$$

$$col(p, v) = v$$

*transitions and flow:*

$$T' \stackrel{\text{def}}{=} \{(t, (\alpha, \beta)) \mid t \in T \wedge (\alpha, \beta) \in \gamma(t)\}$$

$$\bullet(t, (\alpha, \beta)) \stackrel{\text{def}}{=} \{(p, \alpha(p)) \mid p \in \bullet t \cap P\} \cup (\bullet t \cap P^\pm)$$

$$(t, (\alpha, \beta))^\bullet \stackrel{\text{def}}{=} \{(p, \beta(p)) \mid p \in t^\bullet \cap P\} \cup (t^\bullet \cap P^\pm)$$

*initial marking:*

$$\forall (p, v) \in P^\pm \times V, \quad M^{0'}(p, v) = M^0(p, v)$$

$$\forall p' = (p, v) \in P', \quad \forall v' \in V, \quad M^{0'}(p', v') = \begin{cases} M^0(p, v) & \text{if } v = v' \\ 0 & \text{otherwise} \end{cases}$$

*firing modes: for all  $t' = (t, (\alpha, \beta)) \in T'$*

$$\forall p' = (p, v) \in P', \quad \begin{cases} \alpha'_{t'}(p') \stackrel{\text{def}}{=} v & \text{iff } p' \in \bullet t' \\ \beta'_{t'}(p') \stackrel{\text{def}}{=} v & \text{iff } p' \in t'^\bullet \end{cases}$$

$$\forall p \in P^\pm, \quad \alpha'_{t'}(p) \stackrel{\text{def}}{=} \alpha(p) \text{ and } \beta'_{t'}(p) \stackrel{\text{def}}{=} \beta(p)$$

*labels and morphism:*

$\Lambda$  is the label set of  $N$

$$\forall p \in P^\pm, \quad \chi_N(p) \stackrel{\text{def}}{=} p \text{ and } \lambda'(p) \stackrel{\text{def}}{=} \lambda(p)$$

$$\forall p' = (p, v) \in P', \quad \chi_N(p') \stackrel{\text{def}}{=} p \text{ and } \lambda'(p') \stackrel{\text{def}}{=} \lambda(p)$$

$$\forall t' = (t, (\alpha, \beta)) \in T', \quad \chi_N(t') \stackrel{\text{def}}{=} t$$

**Proposition 1.** *The expansion of a colored puzzle net yields an expanded net, and the mapping  $\chi_N : Exp(N) \rightarrow N$  is a (compression) morphism of colored puzzle nets.*

*Proof.* We have to show that  $Exp(N)$  is an expanded net. The only non-trivial part here concerns the condition about the colors in internal places (second item of Definition 4): it is ensured by the definition of the internal places  $P'$  through the pre- and post-sets of the expanded transitions.

Checking that  $\chi_N$  is a morphism from  $Exp(N)$  to  $N$  is straightforward.  $\square$

**Theorem 3 (expansion).** *The Exp functor establishes a coreflection between the category of colored puzzle nets, and the full subcategory of expanded nets.*

*Proof sketch.* We have to prove the universal property of each expanded net  $Exp(N)$ , associated to its compression morphism  $\chi_N$ , i.e. for every morphism  $\phi$  from an expanded net  $N'$  to a puzzle net  $N$ , there exists a unique morphism  $\psi$  from  $N'$  to  $Exp(N)$  such that  $\phi = \chi_N \circ \psi$ . If such a morphism  $\psi$  exists, this latter condition imposes the following definition:

- $\psi(x) = * \iff \phi(x) = *$
- $\phi(p) \in P^\pm \implies \psi(p) \stackrel{\text{def}}{=} \phi(p)$
- $\phi(p) \in P \implies \psi(p) \stackrel{\text{def}}{=} (\phi(p), col(p))$
- $\psi(t) \stackrel{\text{def}}{=} (\phi(t), (\alpha_t \circ \phi_{|\bullet}^{-1}, \beta_t \circ \phi_{|\bullet}^{-1}))$

It remains to show that  $\psi$  does satisfy the conditions for being a morphism from  $N'$  to  $Exp(N)$ , and that  $\phi = \chi \circ \psi$ .  $\square$

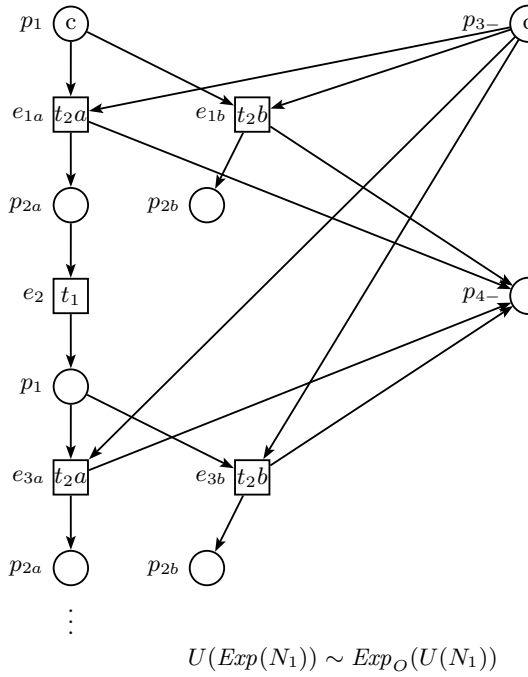
A nice consequence of this coreflection is that expansion and product commute. One has first that there exists a product in the subcategory of expanded nets, defined by  $N_1 \times_E N_2 = Exp(N_1 \times N_2)$ , where  $N_1 \times N_2$  is the product in the sense of colored puzzle nets. Notice that the expansion operator in the right hand side term is necessary in order to expand as well the interface places that become internal after the standard product of colored nets. Now, since products are special cases of categorical limits (theorem 2), and given that limits are preserved by functors that have a left adjoint, one has:

$$Exp(N_1 \times N_2) \sim Exp(N_1) \times_E Exp(N_2)$$

This relation would be tedious to prove directly, so it is interesting to obtain it by standard structural derivations (that exactly reproduce those of [24] in their structure).

## 4 Symbolic Unfolding and Its Properties

Unfoldings provide a compact data structure to encode sets of runs of a Petri net in a true concurrency semantics. By their ability to avoid the combinatorial explosion due to the interleaving of concurrent events, they are particularly suited to analyse properties of distributed systems. And they have indeed been used in this sense, to check the absence of deadlocks, or for reachability analysis. Unfoldings have been defined for ordinary (low-level) safe nets, and more generally for semi-weighted nets. Some authors have extended this construction



**Fig. 3.** The expanded unfolding of the colored puzzle net  $N_1$  of Figure 1

to colored (or high-level) nets by first performing what is called an expansion in this paper, and then applying a standard unfolding procedure, as illustrated in Fig. 3. This is the approach of [17,16] for a model of high-level Petri nets called *M-nets* [7].

In this paper, we propose to go further in this direction, and define directly the unfolding of a colored puzzle net as a *symbolic* unfolding, that is as some form of “colored puzzle branching process.” The principles are the same as in [11] or [9], where we also advocated the interest of symbolic unfoldings for the diagnosis of distributed systems. We then study the relations between symbolic unfolding and expansion. The main contribution of this section is the derivation of a factorization property of symbolic unfoldings, as it was already derived by Winskel [24] for ordinary unfoldings. Namely, the symbolic unfolding of a product of puzzle nets is the product (in a specific sense) of the symbolic unfoldings of the components. This derivation follows the same principles as for the expansion, by producing an adequate coreflection between categories.

*Generic Executions for a Generic Model.* In high-level processes we definitely want to benefit from the generic aspects of the colored puzzle net that we are considering. Indeed in a colored puzzle net, if several states share the same marking (that is the tokens are in the same places but do not carry the same values), we can view these states as instances of a generic family of states. Similarly each



transition is a generic representation of a family of actions, that differ only by the values/colors of the tokens that are consumed and created. And we consider that grouping several states into a generic state or several actions into a high-level transition, results from a choice that was done when the system was modeled.

With respect to this choice, we can identify families of executions of a colored puzzle net  $N$  based on the generic aspects related to its places and transitions. To do this, our approach is based on the fact that each execution of  $N$  can be mapped to an execution of the underlying low-level Petri net obtained by simply removing the colors and the guards.

**Definition 6 (symbolic occurrence net).** A symbolic occurrence net is a colored puzzle net  $O \stackrel{\text{def}}{=} (B, P^+, P^-, E, \text{pre}, \text{post}, \Lambda, \lambda, \gamma, M^0)$  where the internal places, denoted  $B$  here, are called conditions and the transitions, denoted  $E$  here, are called events, which satisfies:

- $\rightarrow^+$  is acyclic, where  $\rightarrow$  denotes the causality relation, defined as  $(e_1 \rightarrow e_2) \stackrel{\text{def}}{\iff} (e_1 \bullet \cap \bullet e_2 \cap B \neq \emptyset)$ . Notice that the interface places induce no causality, since they are not “unfolded”.
- $\forall b \in B$ 

$$\begin{cases} \sum_{v \in V} M^0(b, v) = 0 \wedge \exists! e \in E \ b \in e \bullet & (\text{then this } e \text{ is denoted } \bullet b) \\ \vee \sum_{v \in V} M^0(b, v) = 1 \wedge \nexists e \in E \ b \in e \bullet & (\text{then we define } \bullet b \stackrel{\text{def}}{=} \perp) \end{cases}$$
- $\forall e \in E$ 

$$\begin{cases} [e] \stackrel{\text{def}}{=} \{f \in E \mid f \rightarrow^* e\} \text{ is finite} \\ \nexists e_1, e_2 \in [e] \ e_1 \neq e_2 \wedge \bullet e_1 \cap \bullet e_2 \cap B \neq \emptyset \\ \text{valid\_colorings}([e]) \neq \emptyset \end{cases}$$

where, for every set  $F$  of events,  $\text{valid\_colorings}(F)$  denotes the set of colorings  $\text{Col} : (\bullet F \cup F \bullet) \cap B \rightarrow V$  of the input and output conditions of the events in  $F$ , that are compatible with the firing modes of these events and with the color of the tokens in the initial conditions:

$$\begin{cases} \forall e \in F \ \exists (\alpha, \beta) \in \gamma(e) \ (\alpha|_{\bullet e \cap B}, \beta|_{e \bullet \cap B}) = (\text{Col}|_{\bullet e \cap B}, \text{Col}|_{e \bullet \cap B}) \\ \forall b, v \ M^0(b, v) = 1 \implies \text{Col}(b) = v. \end{cases}$$

In an occurrence net, places are usually called conditions, and transitions are called events. Concerning conditions, the second point in the definition requires that each of them is created (i.e. immediately preceded) by a unique event, or it is minimal, and marked with a single token. On events, the requirements are standard. The first line expresses the well-foundedness (configurations are finite), and the second line expresses that no node should be in (structural) self-conflict. Or equivalently that there is no immediate conflict in the past of each event.

*Treatment of Interface Places.* Note that the interface places are not treated like the internal places. The idea is that only the behaviour of the component is represented in the occurrence net, and no assumption is made about the components it will be connected to. In particular, the events can freely use tokens from the interface places, considering that they may be filled with any number of tokens of any kind.

*Color Conflict.* Processes of a colored Petri net have to satisfy both structural conditions and conditions imposed by the guards on the possible values for the firing modes. The structural conditions only depend on the underlying low-level process and express:

- the causal dependencies (noted  $\rightarrow^+$ ), which induce a partial ordering on events,
- the structural conflicts, identified by the consumption of a condition by two different events, which implies that these two events cannot occur in the same execution, as well as their successors for the causal relation (conflict is inherited by causality),
- the concurrency relation: when two events are neither causally related nor in conflict, they are said to be concurrent. They can then occur in any order in an execution.

But these structural conditions are not sufficient when we deal with symbolic occurrence nets: a set of events can be made incompatible by the fact that there exists no suitable value for the firing modes of the events in their past, even if they would be concurrent in the underlying low-level process.

We can say that a set  $E$  of events of  $O$  are in *color conflict* if they are not in conflict, but the constraints on the values of the firing modes, coming from the guards of the transitions, prevent the events of  $E$  to appear in the same process of  $N$ .

Observe that in the example of Figure 5 the symbolic unfolding of  $N_0$  is finite because  $e_2$  and  $e_5$  are in color conflict: they impose contradictory constraints on the token in  $p_4$  after  $e_1$  fires:  $e_2$  can fire only if it is  $a$ , but  $e_5$  only if it is  $b$ . Nevertheless, without colors, the unfolding would have been infinite.

Unlike the structural conflict due to the consumption of a single condition by several events, color conflicts are not binary in general. That is, the minimal sets of events that are in conflict may have more than two elements.

In the definition of symbolic occurrence nets, color conflicts are treated in the *valid\_colorings* function, which deals both with colors and with the symbolic aspects. The existence of a valid coloring expresses that, for each event  $e$ , there is a way of coloring the configuration  $\lceil e \rceil$  leading to  $e$  in a coherent manner. This coloring assigns a color to every condition and ensures that these colors are compatible with the firing modes of the events.

We are now equipped to define symbolic unfoldings  $U(N)$  of a colored puzzle net  $N$ . A minor and classical restriction on the structure of a Petri net is necessary in order to define its unfolding: we require that every transition consumes at least one token from an *internal place*. Interface places do not really participate in the unfolding and are not duplicated. Moreover the initial marking must not contain more than one token per place (even if they have different colors).

Remark: when dealing with weighted arcs, another condition is also required: the output arcs of the transitions must be simple, i.e. no transition should produce more than one token per output place. These nets are called “semi-weighted

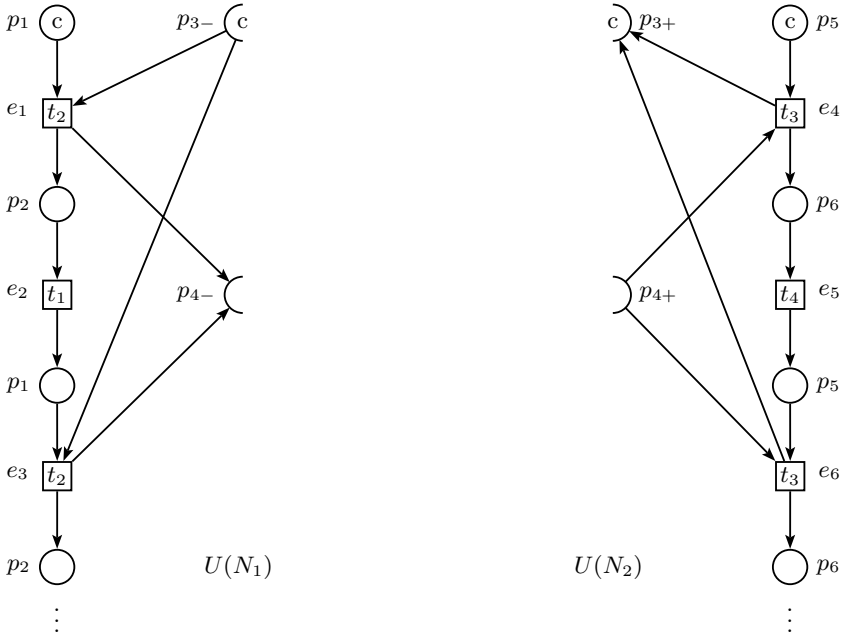


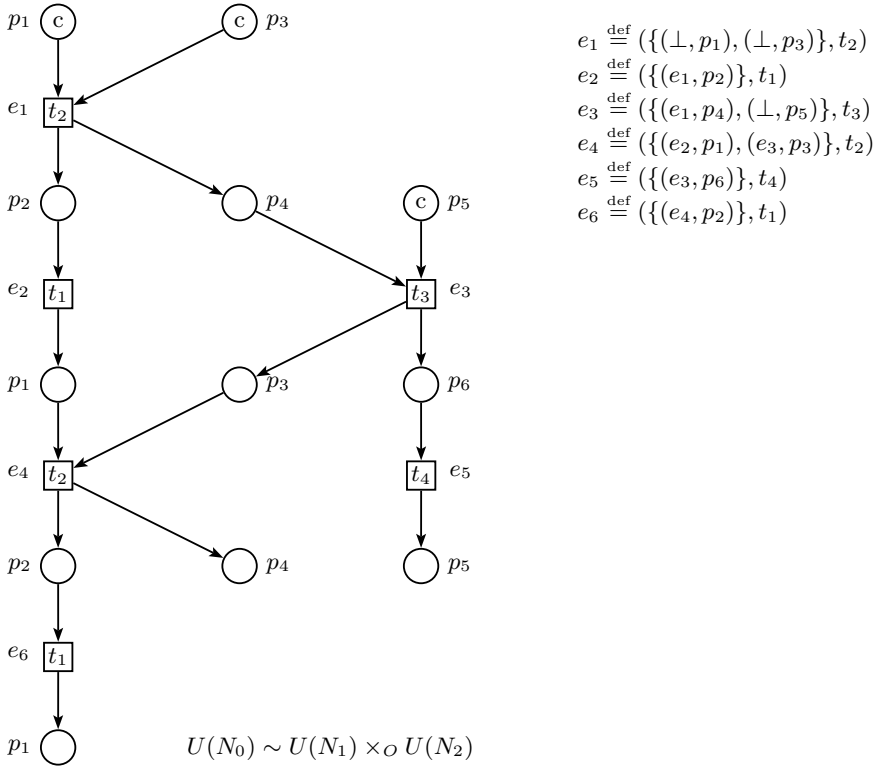
Fig. 4. The symbolic unfolding of each component of the colored puzzle net of Figure 1

nets” in [20,4]. Here we did not consider weighted arcs, so this condition is satisfied by construction.

In our definition of symbolic unfoldings, we use the canonical coding of events and conditions introduced in [12], based on a backward chaining principle. As the unfoldings are occurrence nets, each condition  $b \in B$  is created by a single event  $e$  denoted  $\bullet b$ , if we take the convention that  $\bullet b$  may be either an event of  $E$  or the virtual initial event  $\perp$  when  $b$  represents a token of the initial marking. Moreover, as each event  $e$  of the unfolding of a net  $N$  represents an occurrence of a transition  $t$  of  $N$ , then the output conditions of  $e$  represent the tokens created in the internal places of  $t \cap P$ . Thus each of these conditions is identified by a pair  $(e, p)$  with  $p \in t \cap P$ . Similarly, every event  $e$  of the unfolding is itself a pair  $(C, t)$  where  $t$  is a transition of  $N$ , and  $C \subseteq B$  is the set of conditions that are consumed by  $e$ . As an example, in Figure 5, the coding of the events is written on the right.

The folding morphism  $\phi_N$  from the unfolding  $U(N)$  to the net  $N$  reflects also the correspondence between the events (respectively conditions) of the unfolding and the transitions (respectively places) of the net. It is defined as:

- $\forall e = (C, t) \in E \quad \phi_N(e) = t$ ,
- $\forall b = (e, p) \in B \quad \phi_N(b) \stackrel{\text{def}}{=} p$  and
- $\forall p \in P^\pm \quad \phi_N(p) = p$ , since the interface places are not unfolded.



**Fig. 5.** The symbolic unfolding of the colored puzzle net  $N_0 = N_1 \times N_2$  of Figure 1

**Definition 7 (symbolic unfolding).** Let  $N$  be a colored puzzle net such that

- $\forall p \in P \quad \sum_{v \in V} M^0((p, v)) \leq 1$  and
- $\forall t \in T \quad \bullet t \cap P \neq \emptyset$ .

We define its symbolic unfolding

$$U(N) \stackrel{\text{def}}{=} (B, P^+, P^-, E, pre_U, post_U, \Lambda, \lambda_U, \gamma_U, M_U^0)$$

as follows: ( $B$  and  $E$  are defined inductively)

1.  $P^+$  and  $P^-$  are the sets of interface places of  $N$
2. initial conditions:  
 $\perp \bullet \subseteq B$ , with  $\perp \bullet \stackrel{\text{def}}{=} \{(\perp, p) \mid p \in P, \exists v \in V, M^0(p, v) = 1\}$
3. initial marking:  
 $\forall p \in P^\pm \quad \forall v \in V \quad M_U^0(p, v) \stackrel{\text{def}}{=} M^0(p, v)$   
 $\forall (\perp, p) \in \perp \bullet \quad \forall v \in V \quad M_U^0((\perp, p), v) \stackrel{\text{def}}{=} M^0(p, v)$   
 $\forall b \in B \setminus \perp \bullet \quad \forall v \in V \quad M_U^0(b, v) \stackrel{\text{def}}{=} 0$

4. *input and output of an event:*

$$\forall e = (C, t) \in E \quad \begin{cases} \bullet e \stackrel{\text{def}}{=} C \cup (\bullet t \cap P^\pm) & \text{and} \\ e^\bullet \stackrel{\text{def}}{=} \{(e, p) \mid p \in t^\bullet \cap P\} \cup (t^\bullet \cap P^\pm) \end{cases}$$

5. *firing modes (only those that are compatible with a valid coloring of  $\lceil e \rceil$ ):*

$$\forall e = (C, t) \in E \quad \begin{aligned} \gamma(e) \stackrel{\text{def}}{=} & \{(\alpha \circ \phi_{N|\bullet e}, \beta \circ \phi_{N|e^\bullet}) \mid (\alpha, \beta) \in \gamma(t) \wedge \\ & \exists \text{Col} \in \text{valid\_colorings}(\lceil e \rceil) \quad \text{Col}|_C = \alpha \circ \phi_{N|C}\} \end{aligned}$$

6. *insertion of new events:*

$$\forall e = (C, t) \in 2^B \times T, \quad e \in E \quad \text{iff} \quad \begin{cases} \phi_{N|C} \text{ is a bijection from } C \text{ to } \bullet t \cap P \\ \nexists e_1, e_2 \in \lceil e \rceil \quad e_1 \neq e_2 \wedge \bullet e_1 \cap \bullet e_2 \cap B = \emptyset \\ \exists (\alpha, \beta) \in \gamma(e) \quad \exists \text{Col} \in \text{valid\_colorings}(\lceil e \rceil) \quad \text{Col}|_C = \alpha|_C \end{cases}$$

7. *insertion of new conditions created by an event:*

$$\forall e \in E \quad e^\bullet \setminus P^\pm \subseteq B$$

8. *labels:*

$\Lambda$  is the set of labels of  $N$ ;

$$\forall x \in B \cup P^\pm \quad \lambda_U(x) = \lambda(\phi_N(x))$$

**Proposition 2.**  $U(N)$  is a symbolic occurrence net, and the folding  $\phi_N : U(N) \rightarrow N$  is a morphism.

See the proof in [\[8\]](#).

**Theorem 4 (symbolic unfolding).** *The  $U$  functor establishes a coreflection between the category of (unfoldable) colored puzzle nets and the full subcategory of symbolic occurrence nets.*

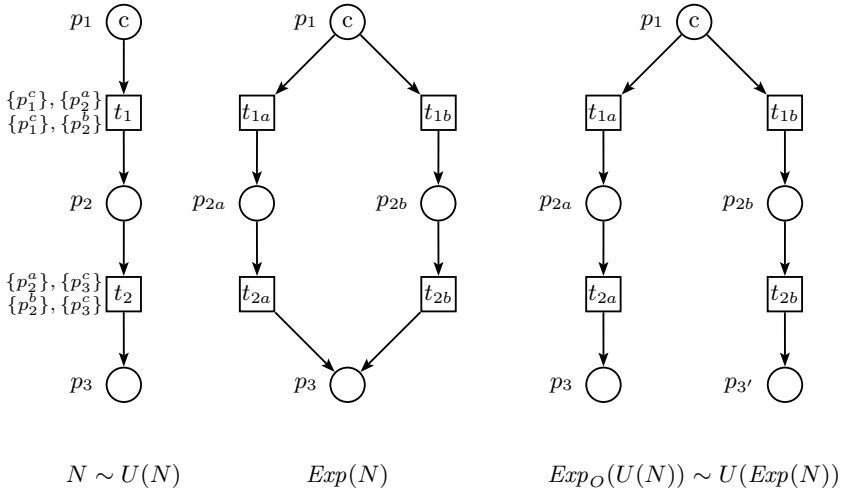
*Proof sketch.* We have to prove the universal property of symbolic unfoldings: Let  $\phi$  be a morphism from a symbolic occurrence net  $O$  to an unfoldable net  $N$ . There exists a unique morphism  $\psi : O \rightarrow U(N)$  such that  $\phi = \phi_N \circ \psi$ . We prove easily that if  $\psi$  exists, it is unique and defined as:

$$\begin{aligned} - \phi(x) \in P^\pm & \implies \psi(x) \stackrel{\text{def}}{=} \phi(x) \\ - \psi(x) = * & \quad \text{iff} \quad \phi(x) = * \\ - \psi(e) & \stackrel{\text{def}}{=} (\psi(\bullet e) \setminus P^\pm, \phi(e)) \text{ if } \phi(e) \neq * \\ - \phi(b) \in P & \implies \begin{cases} \psi(b) \stackrel{\text{def}}{=} (\perp, \phi(b)) & \text{if } \bullet b = \perp \\ \psi(b) \stackrel{\text{def}}{=} (\psi(\bullet b), \phi(b)) & \text{otherwise} \end{cases} \end{aligned}$$

It remains to show that  $\psi$  is a morphism from  $O$  to  $U_N$ . The difficult part is to show that  $\psi$  maps the events of  $O$  to valid events of  $U(N)$ .  $\square$

By the same arguments as for the expansion, one derives immediately the existence of a product in the subcategory of symbolic occurrence nets, given by  $O_1 \times_O O_2 \stackrel{\text{def}}{=} U(O_1 \times O_2)$ . And, again, product is preserved by the symbolic unfolding functor:

$$U(N_1 \times N_2) \sim U(N_1) \times_O U(N_2)$$



**Fig. 6.** A colored Petri net  $N$ , which is isomorphic to its unfolding  $U(N)$ , the expansion  $Exp(N)$  of  $N$  and the expansion  $Exp_O(U(N))$  of the unfolding

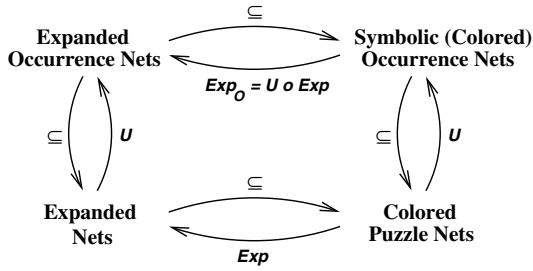
which was the announced result. This is illustrated in Figure 5. Notice that the factored form on the right hand side is by nature more compact than the symbolic unfolding of the product, since interface places between  $N_1$  and  $N_2$  are not expanded.

### 4.1 Expanded Unfoldings and Expansion of Colored Occurrence Nets

Remark that the symbolic unfolding of an expanded puzzle net is an expanded occurrence net. Actually, when applied to expanded puzzle nets, our definition of symbolic unfolding matches the usual definition of unfoldings for low-level Petri nets [24] (up to interface places). This object forms a coreflection from expanded nets to expanded occurrence nets, and we call it the expanded unfolding.

We are looking for a relation between the symbolic unfolding and the expanded unfolding of a colored puzzle net. The idea is that expanding the symbolic unfolding should yield the expanded unfolding. But actually the expansion of an occurrence net is not an occurrence net in general. This fact is illustrated in Figure 6, where transition  $t_1$  of  $N$  produces either a token of color  $a$  or  $b$  in place  $p_2$ , and transition  $t_2$  consumes it anyway and produces a token of color  $c$  in place  $p_3$ . If  $U(N)$  is expanded as a net, then the two versions of transition  $t_3$  converge to the same place  $p_3$ , which is not suitable for an occurrence net.

The correct expansion functor  $Exp_O$  for occurrence nets is defined naturally as  $Exp_O(O) = U(Exp(O))$ . Composing the two coreflections built in the previous sections allows one to establish one more between the category of symbolic occurrence nets and the category of expanded occurrence nets, the former being



**Fig. 7.** Coreflections between categories derived from colored puzzle nets by symbolic unfolding and by expansion

viewed as a subcategory of colored puzzle nets (see Fig. 7). In addition, one has that the expanded unfolding  $U(Exp(N))$  is isomorphic to the expansion by  $Exp_O$  of its symbolic unfolding  $U(N)$ , that is  $U(Exp(N)) \sim Exp_O(U(N))$ .

All this results in the commutative diagram in Fig. 7 that displays the four categories derived from colored puzzle nets by expansion and by symbolic unfolding. The coreflections illustrated in this figure naturally transport categorical limits. For example, for two symbolic occurrence nets  $O_1, O_2$ , one has  $Exp_O(O_1 \times_O O_2) \sim Exp_O(O_1) \times_{EO} Exp_O(O_2)$  where the product  $\times_{EO}$  in the category of expanded occurrence nets is obtained by applying  $Exp_O$  to the product  $\times_O$  in the category of symbolic occurrence nets.

## 5 Conclusion

We have studied the unfoldings of colored puzzle nets, a formalism of high-level Petri nets using the popular composition mechanism based on shared places. An adequate categorical framework has been proposed for this family of nets, based on run-preserving morphisms. Symbolic unfoldings have been also adapted to colored puzzle nets, and related to previous notions of unfoldings for low-level nets, through the notion of expansion. In this adequate categorical framework, we have also illustrated an important property of the symbolic unfolding operation, namely that it commutes with product. The factorization property of unfoldings forms the basis of distributed processing methods for distributed systems (for example distributed diagnosis). We will now explore the interest of symbolic unfoldings for this purpose.

Let us mention that all derivations are presented for the family of colored puzzle nets, because we are convinced of the practical interest of composing nets via shared places. However, the same results remain valid with more ordinary categories of colored nets, where composition is performed by synchronizing transitions carrying identical labels.

## References

1. Baldan, P., Chatain, T., Haar, S., König, B.: Unfolding-based diagnosis of systems with an evolving topology. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 203–217. Springer, Heidelberg (2008)
2. Baldan, P., Corradini, A., Ehrig, H., Heckel, R., König, B.: Compositional semantics for open Petri nets based on deterministic processes. Technical report, University of Pisa, Tech. Rep. TR-01-21 (2001)
3. Baldan, P., Corradini, A., Ehrig, H., König, B.: Open Petri nets: Non-deterministic processes and compositionality. In: ICGT 2008. LNCS, vol. 5214, pp. 257–273. Springer, Heidelberg (2008)
4. Baldan, P., Corradini, A., Montanari, U.: Contextual Petri nets, asymmetric event structures, and processes. *Information and Computation* 171(1), 1–49 (2001)
5. Best, E., Devillers, R., Koutny, M.: The box algebra = Petri nets + process expressions. *Inf. Comput.* 178(1), 44–100 (2002)
6. Best, E., Fleischhack, H., Fraczak, W., Hopkins, R.P., Klaudel, H., Pelz, E.: A class of composable high level Petri nets with an application to the semantics of  $B(PN)^2$ . In: DeMichelis, G., Díaz, M. (eds.) ICATPN 1995. LNCS, vol. 935, pp. 103–120. Springer, Heidelberg (1995)
7. 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)
8. Chatain, T., Fabre, E.: Factorization properties of symbolic unfoldings of colored Petri nets. Research Report LSV-10-07, Laboratoire Spécification et Vérification, ENS Cachan, France (April 2010)
9. Chatain, T., Jard, C.: Symbolic diagnosis of partially observable concurrent systems. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 326–342. Springer, Heidelberg (2004)
10. Ehrig, H., Hoffmann, K., Gabriel, K., Padberg, J.: Composition and independence of high-level net processes. *Electr. Notes Theor. Comput. Sci.* 242(2), 59–71 (2009)
11. Ehrig, H., Hoffmann, K., Padberg, J., Baldan, P., Heckel, R.: High-level net processes. In: Brauer, W., Ehrig, H., Karhumäki, J., Salomaa, A. (eds.) Formal and Natural Computing. LNCS, vol. 2300, pp. 191–219. Springer, Heidelberg (2002)
12. Engelfriet, J.: Branching processes of Petri nets. *Acta Inf.* 28(6), 575–591 (1991)
13. Fabre, E.: On the construction of pullbacks for safe Petri nets. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 166–180. Springer, Heidelberg (2006)
14. Groote, J.F., Voorhoeve, M.: Operational semantics for Petri net components. *Theor. Comput. Sci.* 379(1-2), 1–19 (2007)
15. Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use. Springer, Heidelberg (1995)
16. Khomenko, V.: Model Checking Based on Prefixes of Petri Net Unfoldings. PhD thesis, School of Computing Science, University of Newcastle upon Tyne (2003)
17. Khomenko, V., Koutny, M.: Branching processes of high-level Petri nets. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 458–472. Springer, Heidelberg (2003)
18. Kindler, E.: A compositional partial order semantics for Petri net components. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 235–252. Springer, Heidelberg (1997)



19. Koutny, M., Best, E.: Operational and denotational semantics for the box algebra. *Theor. Comput. Sci.* 211(1-2), 1–83 (1999)
20. Meseguer, J., Montanari, U., Sassone, V.: On the semantics of place/transition Petri nets. *Mathematical Structures in Computer Science* 7(4), 359–397 (1997)
21. Nielsen, M., Priese, L., Sassone, V.: Characterizing behavioural congruences for Petri nets. In: Lee, I., Smolka, S.A. (eds.) *CONCUR 1995*. LNCS, vol. 962, pp. 175–189. Springer, Heidelberg (1995)
22. Priese, L., Wimmel, H.: A uniform approach to true-concurrency and interleaving semantics for Petri nets. *Theor. Comput. Sci.* 206(1-2), 219–256 (1998)
23. Reisig, W.: Simple composition of nets. In: Franceschinis, G., Wolf, K. (eds.) *PETRI NETS 2009*. LNCS, vol. 5606, pp. 23–42. Springer, Heidelberg (2009)
24. Winskel, G.: Categories of models for concurrency. In: Brookes, S.D., Winskel, G., Roscoe, A.W. (eds.) *Seminar on Concurrency*. LNCS, vol. 197, pp. 246–267. Springer, Heidelberg (1985)

# Forward Analysis for Petri Nets with Name Creation<sup>\*</sup>

Fernando Rosa-Velardo and David de Frutos-Escrig

Dpto. de Sistemas Informáticos y Computación  
Universidad Complutense de Madrid  
{fernandorosa, defrutos}@sip.ucm.es

**Abstract.** Pure names are identifiers with no relation between them, except equality and inequality. In previous works we have extended P/T nets with the capability of creating and managing pure names, obtaining  $\nu$ -APNs and proved that they are strictly well structured (WSTS), so that coverability and boundedness are decidable. Here we use the framework recently developed by Finkel and Goubault-Larrecq for forward analysis for WSTS, in the case of  $\nu$ -APNs, to compute the cover, that gives a good over approximation of the set of reachable markings. We prove that the least complete domain containing the set of markings is effectively representable. Moreover, we prove that in the completion we can compute least upper bounds of simple loops. Therefore, a forward Karp-Miller procedure that computes the cover is applicable. However, we prove that in general the cover is not computable, so that the procedure is non-terminating in general. As a corollary, we obtain the analogous result for Transfer Data nets and Data Nets. Finally, we show that a slight modification of the forward analysis yields decidability of a weak form of boundedness called width-boundedness.

## 1 Introduction

Pure names have been extensively studied in the fields of security and mobility, because they can be used to represent different entities widely used in them. For instance, names can represent communicating channels in  $\pi$ -calculus terms, computing boundaries in the Ambient Calculus or ciphering keys in the spi Calculus [13]. In previous works we have extended P/T nets with a primitive to create fresh names, defining  $\nu$ -APNs. Names are represented as tokens, that are no longer indistinguishable. These tokens can move along the places of the net and be used to restrict the firing of some transitions, imposing for instance that two certain names at the preconditions match.

In [17] we proved that  $\nu$ -APNs are Well Structured Transition Systems (WSTS). For WSTS it is possible to perform a backward analysis that computes the set  $\uparrow Pre^*(\uparrow M)$  [18], the set of predecessors of an upward-closed set

---

<sup>\*</sup> Work partially supported by the Spanish projects DESAFIOS10 TIN2009-14599-C03-01, UCM-BSCH GR58/08/910606 and PROMETIDOS S2009/TIC-1465.

$\uparrow M$ . An effective representation of that set allows us to decide the coverability problem, by checking whether the initial marking  $M_0 \in \uparrow Pre^*(\uparrow M)$ . However, the construction of such sets is extremely expensive, with a non primitive recursive complexity [20].

Very recently, Finkel and Goubault-Larrecq have laid the foundation of a theory supporting forward analysis of WSTS [10,11], computing  $\downarrow Post^*(\downarrow M_0)$ , the so called *cover* of the transition system. The cover provides a good over approximation of the set of reachable states, and its construction is generally more efficient in practice than that of  $\uparrow Pre^*(\uparrow M)$ . However, it is not always possible to obtain an effective representation of the cover [3]. The paper [10] establishes a theory for the completion of *well quasi orders* (wqos), so that we can always represent downward-closed sets by means of their least upper bounds. There it is proved that the least completion of  $X$  (that contains an adequate domain of limits, in the sense of [12]) is the so called ideal completion of  $X$ , or equivalently, the sobrification of  $X$  [14].

We will see here that the ideal completion of the set of markings can be effectively represented by mapping markings to the domain  $\mathcal{MS}(\mathcal{MS}(P))$  of finite multisets of finite multisets of places. For that purpose we introduce the domain of  $\omega$ -markings (analogous to the classical notion of  $\omega$ -markings for P/T nets). In an  $\omega$ -marking, not only some identifiers may appear an unbounded number of times in some places, as happens in classical  $\omega$ -markings, but also an unbounded number of different identifiers may occur in a marking.

Assuming a complete domain (thus containing an adequate domain of limits), a generic Karp-Miller procedure to compute the cover is presented in [11]. This procedure is correct provided the WSTS is  $\infty$ -effective, which intuitively means that we can accelerate simple loops (flat loops, in the sense of [4]). We will see that  $\nu$ -APNs are  $\infty$ -effective when we restrict the non-determinism arising in loops, so that we can apply to them the generic Karp-Miller procedure. Unfortunately, when applied to this kind of systems, the procedure is not guaranteed to terminate. We will see that this is unavoidable, since we can reduce the problem of boundedness for reset nets, which is known to be undecidable [7], to the computation of the cover.

Data nets [16] are Petri nets in which tokens are taken from a linearly ordered and dense domain, and capable of performing whole place operations, such as transfers or resets. Transfer Data Nets is the subclass of Data nets in which no resets are allowed, and Petri Data Nets is the subclass of Data Nets (and of Transfer Data Nets) in which no whole-place operation is allowed. Petri Data nets subsume  $\nu$ -APNs [16], so that as a corollary, there cannot be an algorithm computing (a finite basis of) the cover of a Petri Data net, and therefore neither for a Transfer Data net, thus answering negatively to a question posed in [11].

But even if there is no algorithm for the computation of the cover, we can use a slight modification of the forward Karp-Miller procedure to decide width-boundedness of  $\nu$ -APNs [18,5]. A net is width-bounded if only a bounded number of different names appear in each reachable marking. The paper [5] also establishes the decidability of width-boundedness (called m-boundedness there), but

we claim that the algorithm presented there does not properly work in all the cases. This is because the algorithm stops whenever unboundedness is detected. However, width-unbounded nets may be bounded or not, so that we need to further explore the reachability graph to decide width-boundedness. For that purpose, we need ways to finitely represent downward-closed sets of reachable markings, our  $\omega$ -markings. We already knew [18] that width-boundedness is decidable, but we obtain the result here as a simple application of our forward analysis.

The rest of the paper is structured as follows. Section 2 introduces our notations and some basic concepts. In Section 3 we present  $\nu$ -APNs. In Section 4 we show how  $\nu$ -APNs fit in the general framework for forward analysis of WSTS in [10,11]. Section 5 contains our main results: the viability of a forward Karp-Miller procedure for  $\nu$ -APNs, non-computability of the cover and decidability of width-boundedness. Finally, Section 6 presents our conclusions and some directions for further work.

## 2 Preliminaries

**wqos, dcpos.** A quasi order  $\leq$  is a reflexive and transitive binary relation on a set  $X$ . A partial order is an antisymmetric quasi order. A poset is a set endowed with a partial order. We write  $a < b$  if  $a \leq b$  and  $b \not\leq a$ . A quasi order is simply said well (wqo) [9], if for every infinite sequence  $a_0, a_1, \dots$  there are  $i$  and  $j$  with  $i < j$  such that  $a_i \leq a_j$ . Equivalently, an order is a wqo if every sequence has an increasing subsequence.

The downward closure  $\downarrow E$  of  $E \subseteq X$  is  $\{y \in X \mid y \leq x \text{ for some } x \in E\}$ . A set is downward closed iff  $\downarrow E = E$ . A basis of a downward closed set  $E$  is a set  $A$  such that  $\downarrow A = E$ . An element  $x \in X$  is an upper bound of  $E$  if  $y \leq x$  for all  $y \in E$ . We write  $\text{lub}(E)$  to denote the least upper bound of  $E$ , when it exists. An element  $x \in E$  is maximal if  $x = y$  whenever  $x \leq y \in E$ ;  $\text{Max}E$  is the set of maximal elements of  $E$ . A subset  $D$  of  $X$  is said to be directed if  $\text{lub}(\{x, y\})$  exists for all  $x, y \in D$ . A poset is *directed complete* (dcpo) if every directed subset has a least upper bound. For an arbitrary subset  $E$ ,  $\text{Lub}(E) = \{\text{lub}(D) \mid D \text{ directed, } D \subseteq E\}$ . The set  $\text{Lub}(E)$  can be thought of as  $E$  together with all its limits. For a dcpo  $X$ , we write  $x \ll y$  whenever  $y \leq \text{lub}(D)$  implies  $x \leq z$  for some  $z \in D$ , for all directed subset  $D$ .  $X$  is continuous if for all  $x \in X$ ,  $x = \text{lub}\{y \in X \mid y \ll x\}$ .

**WSTS.** A labelled transition system is a tuple  $N = (X, \rightarrow, \text{Act})$  with a set  $X$  of states,  $\text{Act}$  a set of actions and a transition relation  $\rightarrow = \bigcup_{a \in \text{Act}} \overset{a}{\rightarrow}$ , with  $\overset{a}{\rightarrow} \subseteq X \times X$ . We denote by  $\overset{a}{\rightarrow}^*$  (resp.  $\rightarrow^*$ ) the reflexive and transitive closure of  $\overset{a}{\rightarrow}$  (resp.  $\rightarrow$ ).  $\text{Post}_{a,N}(M)$  (or just  $\text{Post}_a(M)$ ) is the set  $\{M' \mid M \overset{a}{\rightarrow} M'\}$  of immediate  $a$ -successors of  $M$ .  $\text{Post}^*(M) = \{M' \mid M \rightarrow^* M'\}$  is the set of reachable states. Both  $\text{Post}_a$  and  $\text{Post}^*$  are extended pointwise to sets of states. A Well Structured Transition System (WSTS) is a tuple  $N = (X, \rightarrow, \text{Act}, \leq)$ , where  $(X, \rightarrow, \text{Act})$  is a labelled transition system, and  $(X, \leq)$  is a wqo, satisfying

the following monotonicity condition<sup>1</sup>:  $M_1 \geq M_2 \xrightarrow{\alpha} M'_2$  implies the existence of  $M'_1$  such that  $M_1 \xrightarrow{\alpha} M'_1 \geq M'_2$ . Given a state  $M$ , the *cover* of  $M$  is the set  $\downarrow Post^*(M)$  (or equivalently,  $\downarrow Post^*(\downarrow M)$  because of monotonicity), and we will denote it by  $Cover_N(M)$  (or just  $Cover(M)$  if there is no confusion). Given an initial state  $M_0$ , the cover of  $N$  is the cover of  $M_0$ .  $N$  is said to be *effective* if  $Post_a(M)$  is finite and computable for all  $M$ , and  $\leq$  is decidable. A WSTS  $(X, \rightarrow, Act, \leq)$  is complete whenever  $(X, \leq)$  is a continuous dcpo and for every  $a \in Act$ ,  $Post_a(Lub(E)) = Lub(Post_a(E))$  for every set  $E$ .

An ideal is a downward closed directed subset. The ideal completion  $\overline{X}$  of a wqo  $X$  is the set of ideals of  $X$ , ordered by inclusion. Given a WSTS  $N = (X, \rightarrow, Act, \leq)$ , the ideal completion of  $N$  is the transition system  $\overline{N} = (\overline{X}, \mapsto, Act)$ , where  $F \mapsto F' = \downarrow \{s' \mid s \xrightarrow{\alpha} s', s \in F\}$ .  $(\overline{X}, \subseteq)$  is a continuous dcpo. However,  $\overline{N}$  is not a WSTS in general. A wqo is an  $\omega^2$ -wqo if it does not contain the Rado's structure, and an  $\omega^2$ -WSTS is a WSTS with an underlying  $\omega^2$ -wqo [15]. Then,  $\overline{N}$  is a WSTS iff  $N$  is a  $\omega^2$ -WSTS [11].

**Multisets.** Given an arbitrary set  $A$ , we will denote by  $\mathcal{MS}(A)$  the set of finite multisets of  $A$ , that is, the mappings  $m : A \rightarrow \mathbb{N}$ . When needed, we identify each set with the multiset defined by its characteristic function, and use set notation for multisets when convenient. We denote by  $S(m)$  the support of  $m$ , that is, the set  $\{a \in A \mid m(a) > 0\}$  and by  $|m| = \sum_{a \in S(m)} m(a)$  the cardinality of  $m$ . Given

two multisets  $m_1, m_2 \in \mathcal{MS}(A)$  we denote by  $m_1 + m_2$  the multiset defined by  $(m_1 + m_2)(a) = m_1(a) + m_2(a)$ . We will write  $m_1 \subseteq m_2$  if  $m_1(a) \leq m_2(a)$  for every  $a \in A$ . Then, we can define  $m_2 - m_1$ , taking  $(m_2 - m_1)(a) = m_2(a) - m_1(a)$ . We will denote by  $\emptyset \in \mathcal{MS}(A)$  the empty multiset. If  $f : A \rightarrow B$  and  $m \in \mathcal{MS}(A)$ , we define  $f(m) \in \mathcal{MS}(B)$  by  $f(m)(b) = \sum_{f(a)=b} m(a)$ .

Every partial order  $\leq$  defined over  $A$  induces a partial order  $\sqsubseteq$  in the set  $\mathcal{MS}(A)$ , given by  $\{a_1, \dots, a_n\} \sqsubseteq \{b_1, \dots, b_m\}$  if there is an injective function  $\iota : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$  such that  $a_i \leq b_{\iota(i)}$  for all  $i$ . If we do not demand  $\iota$  to be injective we obtain the powerdomain order  $\leq_{\exists}^{\forall}$ . We write  $\sqsubseteq_{\iota}$  and  $\leq_{\iota}^{\forall}$  to stress the use of the mapping  $\iota$ . It is well known that if  $\leq$  is a wqo then so is  $\sqsubseteq$ .

### 3 $\nu$ -APNs

In this section we present  $\nu$ -APNs; the reader is referred<sup>2</sup> to [19] for more details. In  $\nu$ -APNs names can be created, communicated and matched. We can use this mechanism to deal with authentication issues [17], correlation or instance isolation [6]. We formalize name management by replacing ordinary tokens by distinguishable tokens. We fix a set  $Id$  of names, that can be carried by tokens of any  $\nu$ -APN. In order to handle these colors, we need matching variables labelling

<sup>1</sup> Different monotonicity notions are considered in [9].

<sup>2</sup> We present here a more general version, that allows weights in arcs and check for inequality. The results in [17,19,18] can be easily transferred to this extended version.

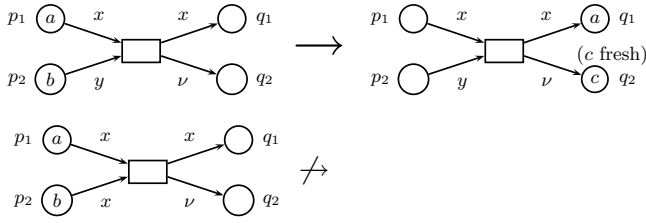


Fig. 1. Two simple  $\nu$ -APN

the arcs of the nets, taken from a fixed set  $Var$ . Moreover, we add a primitive capable of creating new names, formalized by means of special variables in a set  $\Upsilon \subset Var$ , ranged by  $\nu, \nu_1, \dots$ , that can only be instantiated to fresh names.

As an example, the net in the top of Fig. 1 is a simple  $\nu$ -APN with a single transition. When fired, it moves one token from  $p_1$  to  $q_1$  (because of variable  $x$  labelling both arcs), removes a token from  $p_2$  (variable  $y$  does not appear in any outgoing arc) and a new name is created in  $q_2$  (because of variable  $\nu$ ). Instead, the net in the bottom of Fig. 1 uses the same variable  $x$  to label the two arcs incoming its only transition. In that case, the transition must take two tokens carrying the same name from  $p_1$  and  $p_2$ , so that the transition is not enabled.

**Definition 1.** A  $\nu$ -APN is a tuple  $N = (P, T, F)$ , where  $P$  and  $T$  are finite disjoint sets,  $F : (P \times T) \cup (T \times P) \rightarrow \mathcal{MS}(Var)$  is such that for every  $t \in T$ ,  $\Upsilon \cap pre(t) = \emptyset$  and  $post(t) \setminus \Upsilon \subseteq pre(t)$ , where  $pre(t) = \bigcup_{p \in P} S(F(p, t))$  and  $post(t) = \bigcup_{p \in P} S(F(t, p))$ .

The set of pairs  $(x, y)$  such that  $F(x, y) \neq \emptyset$  defines the set of arcs of  $N$ . We also take  $Var(t) = pre(t) \cup post(t)$ ,  $fVar(t) = Var(t) \cap \Upsilon$  and  $nfVar(t) = Var(t) \setminus fVar(t)$ . To avoid tedious definitions, along the paper we will consider a fixed  $\nu$ -APN  $N = (P, T, F)$ .

**Definition 2.** A marking of  $N$  is a function  $M : P \rightarrow \mathcal{MS}(Id)$ . We denote by  $Id(M)$  the set of names in  $M$ , that is,  $Id(M) = \bigcup_{p \in P} S(M(p))$ .

Like for other classes of higher-order nets, transitions are fired with respect to a mode, that chooses which tokens are taken from the preconditions and which are put in the postconditions. Given a transition  $t$  of  $N$ , a mode for  $t$  is an injection  $\sigma : Var(t) \rightarrow Id$  that instantiates each variable to a different identifier. Thus, by using the same variable we force the equality of names taken from preconditions, and because modes are injections, we also check the inequality of names by using different variables. We will use  $\sigma, \sigma', \sigma_1 \dots$  to range over modes.

**Definition 3.** Let  $M$  be a marking,  $t$  a transition and  $\sigma$  a mode for  $t$ . We say  $t$  is enabled with mode  $\sigma$  if for all  $p \in P$ ,  $\sigma(F(p, t)) \subseteq M(p)$  and  $\sigma(\nu) \notin Id(M)$  for all  $\nu \in fVar(t)$ . The reached state after the firing of  $t$  with mode  $\sigma$  is the marking  $M'$ , given by  $M'(p) = (M(p) - \sigma(F(p, t))) + \sigma(F(t, p))$  for all  $p \in P$ .

In the definition of firing we demand that  $\sigma(\nu) \notin Id(M)$ , for every special variable  $\nu$ , that is, that every such  $\nu$  is instantiated to a different fresh name, not in the current marking. Moreover (and unlike in [19]) we demand modes to be injective, which amounts to being able to check for inequality of names (not only for equality, by using the same variable in different arcs). We will write  $M \xrightarrow{t} M'$ ,  $M \xrightarrow{t(\sigma)} M'$ ,  $M \rightarrow M'$  and  $M \xrightarrow{\tau} M'$  with  $\tau = t_1(\sigma_1) \cdots t_n(\sigma_n)$ , saying that  $\tau$  is a transition sequence, with their obvious meanings.

Let us now define the natural order between markings, that induces the coverability problem in  $\nu$ -APN. We define  $M_1 \sqsubseteq_\alpha M_2$  if there is an injection  $\iota : Id(M_1) \rightarrow Id(M_2)$  such that  $\iota(M_1(p)) \subseteq M_2(p)$ , for all  $p \in P$ . We take  $\equiv_\alpha$  as  $\sqsubseteq_\alpha \cap \alpha \sqsupseteq$  and identify markings up to  $\equiv_\alpha$ , that allows renaming of names. The relation  $\sqsubseteq_\alpha$  is a wqo [17]. We will sometimes write  $M_1 \sqsubseteq_\iota M_2$  to emphasize the use of  $\iota$ .

### 4 Forward Analysis for $\nu$ -APNs

The state space of a P/T net is the set  $\mathbb{N}^k$ . However, that set is not complete. For instance, the increasing chain  $(n)_{n=1}^\infty$  does not have a least upper bound in  $\mathbb{N}$ . For that purpose, the classical Karp-Miller construction for P/T nets works instead with the domain  $(\mathbb{N} \cup \{\omega\})^k$ , which is the completion of  $\mathbb{N}^k$ . In particular, the least upper bound of the previous chain is just  $\omega$ . In general, a generic Karp-Miller procedure needs to work with the completion of the domain of the WSTS, in case it is not already complete.

In this section we build the completion of the transition system defined by a  $\nu$ -APN. In [10] it is proved that the ideal completion<sup>3</sup> of a poset is effective (ideals can be finitely represented, and inclusion is decidable) whenever the poset is built up from some basic data type constructions, among which are finite domains, with any order, and multisets of elements in a domain with effective ideal completion. Let us see that we can build our markings using these two constructions.

The behavior of  $\nu$ -APNs is invariant under  $\equiv_\alpha$  [17]. When working modulo  $\equiv_\alpha$  we can represent markings as multisets of multisets of places, where each multiset represents the projection of the marking over some identifier. For instance, the marking  $M$  given by  $M(p) = \{a\}$  and  $M(q) = \{a, b\}$  can be equivalently represented by the multiset  $\{\{p, q\}, \{q\}\}$  in  $\mathcal{MS}(\mathcal{MS}(P))$ . In general, for a marking  $M$ , its multiset representation is given by  $\{M^a \mid a \in Id(M)\}$ , where  $M^a(p) = M(p)(a)$ . We can also denote the previous multiset by the expression  $pq + q$ , where  $pq$  represents the identifier  $a$ , which is both in  $p$  and in  $q$ , and  $q$  represents the identifier  $b$ , which is only in  $q$ . In the following,  $\sqsubseteq$  will denote the natural order over  $\mathcal{MS}(\mathcal{MS}(P))$  (induced by the equality in  $P$ ).

**Lemma 1.** *Let  $M_1$  and  $M_2$  be two markings, and  $\overline{M}_1$  and  $\overline{M}_2$  their multiset representation. Then we have  $M_1 \sqsubseteq_\alpha M_2$  iff  $\overline{M}_1 \sqsubseteq \overline{M}_2$ .*

<sup>3</sup> Actually, the authors work with the equivalent concept of sobrification.

In particular,  $M_1 \equiv_\alpha M_2$  iff their multiset representations coincide. Since we are interested in the abstract treatment of pure names, our set of configurations will be just the set of finite multisets of finite multisets of places<sup>4</sup>.

Next we define  $\omega$ -markings, the analogous concept of the classical  $\omega$ -markings of P/T nets in the case of  $\nu$ -APN. We use a terminology inspired by the Simple Regular Expressions of [3]. We denote by  $\mathbb{N}_\omega$  the set  $\mathbb{N} \cup \{\omega\}$ , and extend the natural order and the usual arithmetic to  $\mathbb{N}_\omega$ . Next we will consider a fixed enumeration of the places of the net,  $P = \{p_1, \dots, p_n\}$ .

**Definition 4.** A product is an expression  $p_1^{i_1} \cdots p_n^{i_n}$  with  $i_1, \dots, i_n \in \mathbb{N}_\omega$ . A sum is an expression of the form  $E_1 + \dots + E_m$ , where each  $E_i$  is a product. An  $\omega$ -marking is an expression  $\mathcal{A} + \infty(\mathcal{B})$ , with  $\mathcal{A}$  and  $\mathcal{B}$  sums.

Intuitively,  $\omega$ -markings are markings (modulo  $\equiv_\alpha$ ) in which some identifiers may appear an unbounded number of times, and also an unbounded number of different identifiers may appear. Notice that each product corresponds to an ordinary  $\omega$ -marking of a P/T net. For instance, the  $\omega$ -marking  $pq^\omega + \infty(p^\omega)$  represents the marking in which an identifier appears once in  $p$  and infinitely often in  $q$ , and infinitely many other different identifiers appear infinitely often in  $p$ . Clearly, plain markings are a particular class of  $\omega$ -markings, those in which  $\mathcal{B}$  is the empty expression and  $E_i = p_1^{i_1} \cdots p_n^{i_n}$  with  $i_1, \dots, i_n \in \mathbb{N}$  for all  $E_i$  in  $\mathcal{A}$ . Sometimes, for an  $\omega$ -marking  $\mathcal{M} = \mathcal{A} + \infty(\mathcal{B})$  we will refer to  $\mathcal{A}$  as the bounded part of  $\mathcal{M}$  and to  $\mathcal{B}$  as the unbounded part of  $\mathcal{M}$ .

We denote by  $\emptyset$  the empty sum, and we will simply write  $\mathcal{A}$  instead of  $\mathcal{A} + \infty(\emptyset)$  and  $\infty(\mathcal{B})$  instead of  $\emptyset + \infty(\mathcal{B})$ . We will often omit places  $p$  with a null exponent, and expand exponential factors, writing for instance  $qq$  instead of  $p^0 q^2$  (assuming  $P = \{p, q\}$ ).

We define  $|p_1^{i_1} \cdots p_n^{i_n}|_\omega = |\{k \mid i_k = \omega\}|$ , and  $(p_1^{i_1} \cdots p_n^{i_n})^\omega = p_1^{j_1} \cdots p_n^{j_n}$ , where  $j_k = 0$  if  $i_k = 0$ , and  $j_k = \omega$  otherwise (e.g.,  $(ppq^\omega)^\omega = p^\omega q^\omega$ ). Given two products  $E_1 = p_1^{i_1} \cdots p_n^{i_n}$  and  $E_2 = p_1^{j_1} \cdots p_n^{j_n}$  we take  $E_1 \sqsubseteq E_2 \Leftrightarrow i_k \leq j_k$  for all  $k \in \{1, \dots, n\}$ , and we define  $E_1 \oplus E_2 = p_1^{i_1+j_1} \cdots p_n^{i_n+j_n}$ , and whenever  $E_2 \sqsubseteq E_1$ ,  $E_1 \ominus E_2 = p_1^{i_1-j_1} \cdots p_n^{i_n-j_n}$ , provided  $j_k \neq \omega$  for all  $k \in \{1, \dots, n\}$ . Finally,  $(\mathcal{A} + \infty(\mathcal{B})) + (\mathcal{A}' + \infty(\mathcal{B}'))$  is the  $\omega$ -marking  $(\mathcal{A} + \mathcal{A}') + \infty(\mathcal{B} + \mathcal{B}')$ .

Let us now define the order between  $\omega$ -markings, that extends the natural one for markings.

**Definition 5.** Given two  $\omega$ -markings  $\mathcal{M} = E_1 + \dots + E_m + \infty(E_{m+1} + \dots + E_k)$  and  $\mathcal{M}' = E'_1 + \dots + E'_{m'} + \infty(E'_{m'+1} + \dots + E'_{k'})$  we define  $\mathcal{M} \sqsubseteq \mathcal{M}'$  if there is  $\iota : \{1, \dots, k\} \rightarrow \{1, \dots, k'\}$  such that:

- If  $\iota(i) = \iota(j)$  and  $\iota(j) \leq m'$  then  $i = j$  (it is partially injective),
- If  $i > m$  then  $\iota(i) > m'$ ,
- $E_i \sqsubseteq E_{\iota(i)}$  for all  $i \in \{1, \dots, k\}$ .

<sup>4</sup> Notice that  $\mathcal{MS}(P)$  is isomorphic to  $\mathbb{N}^{|P|}$ , so that alternatively we could have considered  $\mathcal{MS}(\mathbb{N}^{|P|})$  instead of  $\mathcal{MS}(\mathcal{MS}(P))$ .



As for multisets, we use a mapping  $\iota$  to specify which product of  $\mathcal{M}'$  is used to bound each product in  $\mathcal{M}$ . Products in the bounded part of  $\mathcal{M}$  can be mapped to products in the bounded or in the unbounded part of  $\mathcal{M}'$ , though products in the unbounded part of  $\mathcal{M}$  can only be mapped to products that are also in the unbounded part of  $\mathcal{M}'$ . Intuitively, infinitely many copies of a product can only be bounded by an infinite number of products. Products in the bounded part of  $\mathcal{M}'$  can only be used once to bound products in  $\mathcal{M}$ , while this is not the case for products in the unbounded part. Alternatively, we could have defined  $\mathcal{A} + \infty(\mathcal{B}) \sqsubseteq \mathcal{A}' + \infty(\mathcal{B}')$  if we can split  $\mathcal{A}$  into  $\mathcal{A}_1$  and  $\mathcal{A}_2$  so that  $\mathcal{A}_1 \sqsubseteq \mathcal{A}'$ ,  $\mathcal{A}_2 \leq_{\exists}^{\forall} \mathcal{B}'$ , and  $\mathcal{B} \leq_{\exists}^{\forall} \mathcal{B}'$ . The products in  $\mathcal{A}_1$  are mapped to the bounded part, while the ones in  $\mathcal{A}_2$  and in  $\mathcal{B}$  are mapped to the unbounded part. Notice that, in this case, we are using the order  $\leq_{\exists}^{\forall}$  since, intuitively, we have infinitely many copies of the products in  $\mathcal{B}'$ , so that we can choose any of them to bound as many sums as needed, so that the mapping needs not be injective. For instance, it holds  $p + q + qq + \infty(q) \sqsubseteq pq + \infty(qq)$  because  $p \sqsubseteq pq$ ,  $q + qq \leq_{\exists}^{\forall} qq$  and  $q \leq_{\exists}^{\forall} qq$ .

We take  $\sqsubseteq$  as  $\sqsubseteq \cap \supseteq$  and identify  $\omega$ -markings up to  $\equiv$ . We take as  $\omega$ -Markings the set of  $\omega$ -markings identified up to  $\equiv$ . As for plain markings, we will also use the notation  $\sqsubseteq_{\iota}$ . When there is no confusion, we will write  $\iota(E_i)$  instead of  $E_{\iota(i)}$ . For instance,  $p + qq + \infty(q) \sqsubseteq_{\iota} p + \infty(qq)$  with  $\iota(p) = p$ ,  $\iota(qq) = qq$  and  $\iota(q) = qq$ . The following equivalences will be used along the rest of the paper.

**Lemma 2.** *If  $E_1 \sqsubseteq E_2$  then  $E_1 + \infty(E_2) \equiv \infty(E_2)$  and  $\infty(E_1 + E_2) \equiv \infty(E_2)$ .*

Thus, for instance we have that  $p + q + \infty(pq) \equiv \infty(pq) \equiv \infty(p + q + pq)$ . Though  $\omega$ -markings can be intuitively seen as markings in which some identifiers appear infinitely often, and in which an infinite number of different identifiers can appear, technically they represent sets of markings, those bounded by them as expressed by their denotations.

**Definition 6.** *The denotation of a product  $E = p_1^{k_1} \dots p_n^{k_n}$  is the set of multisets of places  $\llbracket E \rrbracket = \{A \in \mathcal{MS}(P) \mid A(p_i) \leq k_i \text{ for all } i = 1, \dots, n\}$ . The denotation of a sum  $\mathcal{A} = \sum_{i=1}^m E_i$  is given by  $\llbracket \mathcal{A} \rrbracket = \{\{A_i \mid A_i \in \llbracket E_i \rrbracket, i \in I\} \mid I \subseteq \{1, \dots, m\}\}$ . We define the denotation of an  $\omega$ -marking  $\mathcal{M} = \mathcal{A} + \infty(\mathcal{B})$  as the set of markings  $\llbracket \mathcal{M} \rrbracket = \{M + \sum_{i=1}^k M_i \mid k \geq 0, M \in \llbracket \mathcal{A} \rrbracket, M_i \in \llbracket \mathcal{B} \rrbracket\}$ .*

Take the  $\omega$ -marking  $pq + \infty(qq)$ . The denotation of  $pq$  is the set  $\{\emptyset, p, q, pq\}$ , and  $\llbracket qq \rrbracket = \{\emptyset, q, qq\}$ . Thus,  $\llbracket pq + \infty(qq) \rrbracket$  is the set of markings of the form  $M + \underbrace{q + \dots + q}_{n_1} + \underbrace{qq + \dots + qq}_{n_2}$  with  $n_1, n_2 \geq 0$  and  $M \in \llbracket pq \rrbracket$ . Notice that  $\llbracket \mathcal{M} \rrbracket$

is a downward closed and directed set, that is, an ideal.

**Proposition 1.** *The ideal completion of  $(\mathcal{MS}(\mathcal{MS}(P)), \sqsubseteq)$  can be effectively represented as  $(\omega\text{-Markings}, \sqsubseteq)$ .*

---

<sup>5</sup> Abusing notation, we are considering sums to be multisets of products.

In particular, given two  $\omega$ -markings  $\mathcal{M}_1$  and  $\mathcal{M}_2$  it holds that  $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2 \Leftrightarrow \llbracket \mathcal{M}_1 \rrbracket \subseteq \llbracket \mathcal{M}_2 \rrbracket$ , so that  $(\omega\text{-Markings}, \sqsubseteq)$  is a continuous dcpo.

Now we need to lift the transition relation to the completed domain of  $\omega$ -markings. More precisely, for each  $\omega$ -marking  $\mathcal{M}$  we need to effectively compute the set  $\downarrow \text{Post}(\llbracket \mathcal{M} \rrbracket)$ . First, let us introduce some notations: Given a transition  $t$  and a variable  $x$ , we will denote by  $\text{pre}_t(x)$  the product  $p_1^{i_1} \cdots p_n^{i_n}$ , with  $i_k = F(p_k, t)(x)$ , and  $\text{post}_t(x) = p_1^{i_1} \cdots p_n^{i_n}$ , with  $i_k = F(t, p_k)(x)$ . In particular, the products  $\text{post}_t(\nu)$ , that correspond to the special variables  $\nu \in \mathcal{T}$ , are the “fresh” products created by the transition  $t$ . For instance, the net in the bottom of Fig. 1 satisfies  $\text{pre}_t(x) = p_1 p_2$ ,  $\text{post}_t(x) = q_1$  and  $\text{post}_t(\nu) = q_2$ .

**Definition 7.** Let  $\mathcal{M} = E_1 + \cdots + E_m + \infty(E_{m+1} + \cdots + E_k)$  be an  $\omega$ -marking, and  $t$  a transition. An  $\omega$ -mode for  $t$  is any mapping  $\sigma : \text{nfVar}(t) \rightarrow \mathbb{N}$  such that:

- If  $\sigma(x) = \sigma(y)$  and  $\sigma(y) \leq m$  then  $x = y$ , and
- $\text{pre}_t(x) \sqsubseteq E_{\sigma(x)}$  for all  $x \in \text{Var}(t)$ .

Then we write  $\mathcal{M} \xrightarrow{t(\sigma)} \mathcal{A} + \infty(\mathcal{B})$ , where  $\mathcal{B} = E_{m+1} + \cdots + E_k$  and

$$\mathcal{A} = \sum_{x \in \text{nfVar}(t)} ((E_{\sigma(x)} \ominus \text{pre}_t(x)) \oplus \text{post}_t(x)) + \sum_{i \notin \sigma(\text{Var}(t))} E_i + \sum_{\nu \in \text{fVar}(t)} \text{post}_t(\nu)$$

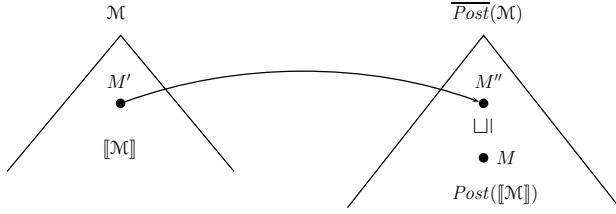
We define  $\overline{\text{Post}}_t(\mathcal{M}) = \{\mathcal{M}' \mid \mathcal{M} \xrightarrow{t(\sigma)} \mathcal{M}' \text{ for some } \sigma\}$ , and extend it pointwise to sets of  $\omega$ -markings.

We will write  $\sigma(x) = E$  to denote that the product  $E$  is used by variable  $x$  in mode  $\sigma$ . For all  $x \in \text{Var}(t)$ , we will write  $\nabla_t(x) = (\sigma(x) \ominus \text{pre}_t(x)) \oplus \text{post}_t(x)$ . Notice that for  $\nu \in \text{fVar}(t)$  then  $\nabla_t(\nu)$  is simply  $\text{post}_t(\nu)$ . We will also write  $\mathcal{M} \xrightarrow{t} \mathcal{M}'$ ,  $\mathcal{M} \rightarrow \mathcal{M}'$ ,  $\mathcal{M} \xrightarrow{\tau} \mathcal{M}'$  and  $\mathcal{M} \rightarrow^* \mathcal{M}'$  as with plain markings, with their obvious meanings. Moreover, if the product  $E$  in  $\mathcal{M}$  evolves to  $E'$  in  $\mathcal{M}'$  we will also write  $E \xrightarrow{t(\sigma)} E'$  or  $E \rightarrow E'$ . Notice that whenever  $\mathcal{M} \rightarrow \mathcal{M}'$  the unbounded part of  $\mathcal{M}$  and  $\mathcal{M}'$  coincide. However, new products may appear in the bounded part of  $\mathcal{M}'$ , like those in the unbounded part of  $\mathcal{M}$  involved in the firing of the transition. For instance, the net in Fig. 2 can fire  $p + \infty(q) \xrightarrow{t_2(\sigma)} p + qq + \infty(q)$  with  $\sigma(x) = q$ . Intuitively, one of the infinitely many names in  $q$  has been chosen, and put twice in  $q$  by the transition.

Let us see that we can use  $\overline{\text{Post}}_t(\mathcal{M})$  to compute  $\downarrow \text{Post}_t(\llbracket \mathcal{M} \rrbracket)$ . For that purpose, we need the following lemma. From now on, we will denote just by  $\llbracket \overline{\text{Post}}_t(\mathcal{M}) \rrbracket$  the set  $\bigcup_{\mathcal{M}' \in \overline{\text{Post}}_t(\mathcal{M})} \llbracket \mathcal{M}' \rrbracket$ .

**Lemma 3.** The following conditions hold:

- If  $M \in \llbracket \mathcal{M} \rrbracket$  and  $M \xrightarrow{t} M'$  then we have  $M' \in \llbracket \overline{\text{Post}}_t(\mathcal{M}) \rrbracket$ .
- If  $M \in \llbracket \overline{\text{Post}}_t(\mathcal{M}) \rrbracket$  then there are  $M' \in \llbracket \mathcal{M} \rrbracket$  and  $M'' \in \llbracket \overline{\text{Post}}_t(\mathcal{M}) \rrbracket$  such that  $M \sqsubseteq M''$  and  $M' \xrightarrow{t} M''$ .



**Fig. 2.** Computation of  $Post_t(\llbracket M \rrbracket)$

The first part of the previous lemma states that  $Post_t(\llbracket M \rrbracket) \subseteq \llbracket \overline{Post_t(M)} \rrbracket$ . For a better insight of the second part, see Fig. 2. Both allow us to prove the following result.

**Proposition 2.**  $\downarrow Post_t(\llbracket M \rrbracket) = \llbracket \overline{Post_t(M)} \rrbracket$

**Corollary 1.** *The completion  $\overline{N}$  of a  $\nu$ -APN  $N$  is an effective complete WSTS.*

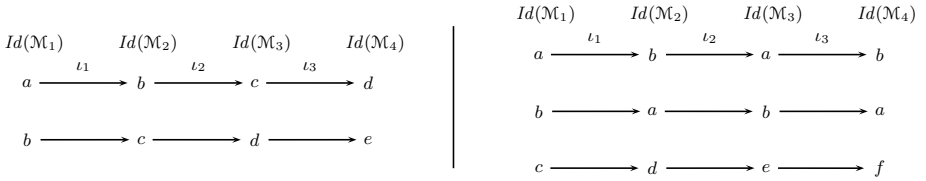
For a complete WSTS, the *clover*  $\llbracket \square \rrbracket$  of a state  $M$  is defined by  $Clover(M) = Max\ Lub(Cover(M))$ . The clover of a state is finite because our order is well. It holds that  $\downarrow Clover(M) = Lub(Cover(M))$ , so that the clover is a finite basis of the cover (together with all the limits). Moreover, if  $\overline{N}$  is the completion of  $N = (X, \rightarrow, \leq)$  then  $Clover_N(M) = Cover_{\overline{N}}(M) \cap X = \downarrow Clover_{\overline{N}}(M) \cap X$ , so that the clover of the completion is a basis of the cover (once we remove the limits by intersecting with  $X$ ).

Now let us see that we can apply a forward Karp-Miller algorithm to compute the clover of  $N$  (although, as we will see, it will not terminate in general). For that purpose, we will need to compute the least upper bounds of all the  $\omega$ -markings produced in a loop, that is, we need to accelerate loops.

## 5 Accelerations

In the previous section we have mostly seen how  $\nu$ -APNs fit in the general framework of  $\llbracket \square \rrbracket$ . In the classic construction of the Karp-Miller tree for P/T nets, every time a transition sequence  $\tau$  such that  $M \xrightarrow{\tau} M'$  with  $M(p) \leq M'(p)$  for all  $p$  and  $M(q) < M'(q)$  for some  $q$ , we know that the transition sequence  $\tau$  can be repeated arbitrarily often, so that the number of tokens in  $q$  can be considered to be unbounded. In other words, we can replace  $M'$  by the least upper bound of the markings obtained by repeating  $\tau$  an arbitrary number of times.

In order to translate the Karp-Miller procedure to  $\nu$ -APNs, we need to prove that the completion of a  $\nu$ -APN is  $\infty$ -effective, meaning that we can compute the least upper bound of the markings obtained by repeating a transition sequence, that is, that we can accelerate loops. In the previous section we have shown how we can effectively represent the completed domains, so that the limit of an increasing chain (and more generally, of a directed set) always exists. However,



**Fig. 3.** Example of construction of the sequences  $(\iota_i)_{i=1}^{\infty}$

the *double infiniteness* in  $\omega$ -markings makes the task of computing those limits a non trivial one. We now specify what will it mean in our setting to repeat a transition sequence.

We will discuss the case in which  $\tau$  is a single transition  $t$ , because the general case would only obscure the presentation. Later we will see how the general case can also be considered. Let us suppose that  $\mathcal{M}_1 \xrightarrow{t(\sigma_1)} \mathcal{M}_2$  and  $\mathcal{M}_1 \sqsubseteq_{\iota_1} \mathcal{M}_2$ . Intuitively, because of monotonicity we can repeat the firing of  $t$  in  $\mathcal{M}_2$ . However, the occurrence of a token  $a$  in  $p$  is bounded by the occurrence of  $\iota_1(a)$  in  $p$ . Therefore, if  $t$  used a token  $a$  because  $\sigma_1(x) = a$  for some variable  $x$ , then now  $t$  must use  $\iota(a)$  instead, thus taking  $\sigma_2(x) = \iota(a)$ . We define the sequences  $(\sigma_i)_{i=1}^{\infty}$ ,  $(\mathcal{M}_i)_{i=1}^{\infty}$  and  $(\iota_i)_{i=1}^{\infty}$  of  $\omega$ -modes,  $\omega$ -markings and mappings, respectively, as follows:

- $\sigma_{i+1}(x) = \iota_i(\sigma_i(x))$ , for  $i \geq 1$ ,
- $\mathcal{M}_i \xrightarrow{t(\sigma_i)} \mathcal{M}_{i+1}$  for  $i \geq 1$ , and
- $\iota_{i+1}(E) = \begin{cases} E' \text{ if } F' \xrightarrow{t(\sigma_i)} E \text{ and } \iota_i(F') \xrightarrow{t(\sigma_{i+1})} E' \text{ for } F' \text{ in } \mathcal{M}_i \\ E \text{ and } E' \text{ in the bounded part,} \\ E \text{ otherwise} \end{cases} \text{ for } i \geq 1.$

$\sigma_{i+1}$  is defined following the previous intuitions: if a variable  $x$  is first instantiated by a product  $E$ , in the next step it is instantiated by  $\iota_i(E)$ .  $\mathcal{M}_{i+1}$  is simply obtained by letting  $\mathcal{M}_i$  evolve with mode  $\sigma_i$ . The definition of the mappings  $\iota_i$  require further explanations. The mappings  $\iota_i$  map products to products, but perhaps their definition is better understood by considering not the products themselves, but the identifier that each product represents. Consider the left handside of the diagram in Fig. 3, where  $a$  is mapped to  $b$  by  $\iota_1$ , and  $b$  is mapped to a fresh identifier  $c$ . The definition of  $\iota_2$  above simply states that now (the product representing)  $b$  is mapped to (the product representing)  $c$ , because  $b$  was mapped to  $c$  by  $\iota_1$ . Accordingly, since  $\iota_1$  mapped  $b$  to a fresh identifier (represented by a product  $E = post_{\iota}(\nu)$  for some  $\nu \in \mathcal{Y}$ ),  $\iota_2$  must map  $c$  to another fresh identifier (which is represented by the same product  $E = post_{\iota}(\nu)$ ). Finally, if  $E$  is in the unbounded part of  $\mathcal{M}_i$  then it is also in the unbounded part of  $\mathcal{M}_{i+1}$ , and  $\iota_{i+1}(E) = E$ .

We will denote by  $t(\sigma_1)_i^k$  the sequence  $t(\sigma_1) \cdot \dots \cdot t(\sigma_k)$ . In general, for a transition sequence  $\tau$  we can define as above the sequences of  $\omega$ -modes,  $\omega$ -markings and mappings. This is because we can always simulate the effect of the firing of a transition sequence using some given modes with the firing of a single transition.

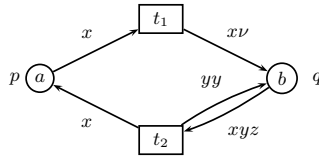


Fig. 4. From transition sequences to transitions

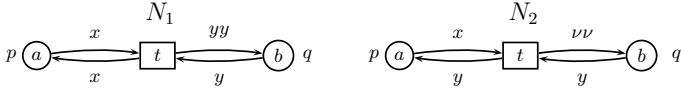


Fig. 5. w-accelerations and d-accelerations

**Lemma 4.** *Let  $\tau$  be a transition sequence of a  $\nu$ -APN  $N = (P, T, F)$ . Then there is a  $\nu$ -APN  $N' = (P, \{\bar{t}\}, F')$  such that  $\mathcal{M}_1 \xrightarrow{\tau} \mathcal{M}_2$  in  $N$  if and only if  $\mathcal{M}_1 \xrightarrow{\bar{t}(\sigma)} \mathcal{M}_2$  in  $N'$  for some mode  $\sigma$  of  $\bar{t}$ .*

We will call  $\tau$ -contraction of  $N$  to the net  $N'$  given by the previous result. We will also write  $\tau_\iota^k$  to denote the transition sequence  $\bar{t}(\sigma)_\iota^k$ , where  $\bar{t}(\sigma)$  is the only transition of its  $\tau$ -contraction. Consider for instance the net in Fig. 4 and the transition sequence  $\tau = t_1(\sigma_1)t_2(\sigma_2)$ , where  $\sigma_1(x) = a$ ,  $\sigma_1(\nu) = c$ ,  $\sigma_2(x) = b$ ,  $\sigma_2(y) = c$  and  $\sigma_2(z) = a$ . The  $\tau$ -contraction of that net is the net  $N_2$  depicted in Fig. 5. Notice that the modes  $\sigma_1$  and  $\sigma_2$  are such that  $\sigma_1(x) = \sigma_2(z)$ . Accordingly, since  $t_1$  puts once  $\sigma_1(x)$  in  $q$ , and  $t_2$  removes  $\sigma_2(z)$  from  $q$ , in  $N_2$  the token  $a$  is neither put nor removed from  $q$ .

We are now ready to define in our setting what it means to accelerate a simple loop. The sequence  $(\mathcal{M}_i)_{i=1}^\infty$  is an increasing sequence, so that the following definition makes sense.

**Definition 8.** *Let  $\bar{N}$  be the completion of a  $\nu$ -APN  $N$ . We say  $\bar{N}$  is  $\infty$ -effective if it is effective and whenever  $\mathcal{M}_1 \xrightarrow{\tau} \mathcal{M}_2$  with  $\mathcal{M}_1 \sqsubseteq_\iota \mathcal{M}_2$  we can compute*

$$acc_\iota(\mathcal{M}_1 \xrightarrow{\tau} \mathcal{M}_2) = lub\{\mathcal{M} \mid \mathcal{M}_1 \xrightarrow{\tau_\iota^n} \mathcal{M}, n > 0\}$$

Let us see that we can compute that least upper bound. In the first place, we can compute the  $\tau$ -contraction of the net, and work with it instead. Therefore, we can always assume that we want to accelerate a single transition. Let us consider the nets  $N_1$  and  $N_2$  in Fig. 5. Notice that both nets can fire the run  $p + q \xrightarrow{t} p + qq$ , and  $p + q \sqsubseteq_\iota p + qq$  with  $\iota(p) = p$  and  $\iota(q) = qq$ . However, the result of an acceleration in both cases is very different: for  $N_1$ , every marking of the form  $p + q^n$  is reachable; for  $N_2$ , every marking  $p + qq + q + \dots + q$  is reachable. Intuitively, the difference between both situations is that in  $N_1$  each product is mapped to itself (the product  $p$  evolves to  $\iota(p) = p$  and the product  $q$  evolves to  $\iota(q) = qq$ ). However, that is not the case for  $N_2$ , where the product  $q$  evolves to  $\iota(p) = p$ . If we consider not products, but the identifiers they represent, then

the difference becomes clearer. In  $N_1$  both  $a$  and  $b$  are mapped to themselves by  $\iota$ , while in  $N_2$ ,  $a$  is mapped to  $b$ , and  $b$  is mapped to a fresh identifier. We formalize the behavior of  $N_1$  in the following definition.

**Definition 9.** We say  $\mathcal{M}_1 \xrightarrow{t(\sigma)} \mathcal{M}_2$  is properly increasing if  $\mathcal{M}_1 \sqsubseteq_\iota \mathcal{M}_2$  and for all products  $E_2$  in  $\mathcal{M}_2$  there are no different products  $E_1$  and  $E'_1$  in the bounded part of  $\mathcal{M}_1$  such that  $E_1 \xrightarrow{t(\sigma)} E_2$  and  $\iota(E'_1) = E_2$ .

The firing  $p + q \xrightarrow{t} p + qq$  is properly increasing in  $N_1$ , but not in  $N_2$ , because there is a product  $p$  in  $p + qq$ , and two different products in  $p + q$ , namely  $p$  and  $q$ , such that  $p$  is mapped to  $p$  by  $\iota_1$  and  $q$  evolves to  $p$ . However, every increasing firing can be unrolled into a properly increasing one. Indeed, consider again the diagrams in Fig. 3. In both parts of the diagrams, there is a natural  $k$  so that each identifier is mapped in  $k$  steps either to itself, or to a fresh identifier. In the left handside, after two steps, both  $a$  and  $b$  are mapped to fresh identifiers. In the right handside, after three steps, both  $a$  and  $b$  are mapped to themselves, but  $c$  is mapped to a fresh identifier. This happens in general, as we will see in the next lemma.

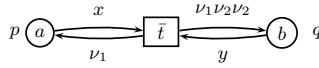
**Lemma 5.** If  $\mathcal{M} \xrightarrow{t(\sigma)} \mathcal{M}'$  and  $\mathcal{M} \sqsubseteq_\iota \mathcal{M}'$  then there is  $k > 0$  such that the firing of the  $\bar{t}(\sigma)_\iota^k$ -contraction of  $N$  is properly increasing.

We call order of  $\iota$ , that we will denote as  $o(\iota)$ , to the natural  $k$  given by the previous result, which can be effectively computed. Moreover, we will write  $\bar{t}(\sigma)$  instead of  $t(\sigma)_{\iota}^{o(\iota)}$ , when it is clear from the context. Clearly,  $acc_\iota(\mathcal{M}_1 \xrightarrow{t(\sigma)} \mathcal{M}_2) = acc_\iota(\mathcal{M}_1 \xrightarrow{t(\sigma)_\iota^k} \mathcal{M})$  for any  $k > 0$  so that, in particular, we can take  $k = o(\iota)$ . Moreover, by Lemma 4 we can work with the  $\bar{t}(\sigma)$ -contraction of the net instead.

As an example, consider the nets in Fig. 5. In  $N_1$  after one step each identifier can be mapped to itself, that is,  $\mathcal{M}_1 \xrightarrow{t(\sigma)} \mathcal{M}_2$  with  $\mathcal{M}_1 \sqsubseteq_\iota \mathcal{M}_2$ ,  $\iota(a) = a$  and  $\iota(b) = b$ , so that  $o(\iota) = 1$ . In  $N_2$  we find the situation in the left of Fig. 3, so that  $o(\iota) = 2$ . Thus, we need to consider the transition sequence  $\tau = t(\sigma)t(\sigma')$ , with  $\sigma'(x) = b$  and  $\sigma'(y) = \sigma(\nu)$ . In turn, in order to compute the acceleration we can consider its  $\tau$ -contraction, depicted in Fig. 6.

**d-acceleration.** Using properly increasing sequences has the advantage that whenever a product  $E_x$  (with  $\sigma(x) = E_x$ ) evolves to some  $E'_x$  in the range of  $\iota$ , then necessarily  $E_x \sqsubseteq E'_x$ . Then, by repeating the firing of  $t$  we will obtain products of the form  $E_x \oplus \Delta_t(x)^k$ , for some *increment*  $\Delta_t(x)$ , with least upper bound  $E_x \oplus \Delta_t(x)^\omega$ . This is the situation for  $N_1$  in Fig. 5 and  $p + q \xrightarrow{t} p + qq$ , that is properly increasing. Using the previous notations,  $E_x = p$  and  $E_y = q$ , so that  $\Delta_t(x) = \emptyset$  and  $\Delta_t(y) = q$ . Therefore,  $acc_\iota(p + q \xrightarrow{t(\sigma)} p + qq) = p + q^\omega$ .

**w-acceleration.** However, in  $N_2$  we cannot apply the previous acceleration. In this case the  $\bar{t}(\sigma)$ -contraction of  $N_2$  is given by the net in Fig. 6. In it, every



**Fig. 6.** Contraction of the net  $N_2$  in Fig. 5

product is mapped to a fresh one, and every marking of the form  $pq+qq+q+\dots+q$  is reachable. If we take  $\Delta_t^t(x) = post_t(\nu_1) \ominus pre_t(x)$  and  $\Delta_t^t(y) = post_t(\nu_2) \ominus pre_t(y)$  then  $acc_t(p + q \xrightarrow{t(\sigma)} p + qq) = pq + qq + \infty(q) \equiv post_t(\nu_1) + post_t(\nu_2) + \infty(\Delta_t^t(x) + \Delta_t^t(y))$ .

A simpler case in which a w-acceleration can be applied appears in the net in Fig. 8. The first firing that takes place is  $p \xrightarrow{t_1} p+q$ , so that  $p \sqsubseteq_t p+q$  with  $\iota(p) = p$ . Notice that there is a fresh product, namely  $q$ , not in the range of  $\iota$ , so that any marking of the form  $p + q + \dots + q$  is reachable, and  $acc_t(p \xrightarrow{t_1} p+q) = p + \infty(q)$ .

Following the previous intuitions, if  $\mathcal{M}_1 \xrightarrow{t(\sigma)} \mathcal{M}_2$  is properly increasing we partition  $nfVar(t)$  as follows:

$$\begin{aligned} V_{un} &= \{x \in nfVar(t) \mid \sigma(x) \text{ in the unbounded part}\}, \\ V_d &= \{x \in nfVar(t) \mid \sigma(x) \xrightarrow{t(\sigma)} \iota(\sigma(x))\}, \\ V_w^\nu &= \{x \in nfVar(t) \mid \iota(x) = post_t(\nu_x) \text{ for } \nu_x \in fVar(t)\}, \\ V_w^{un} &= \{x \in nfVar(t) \mid \iota(\sigma(x)) = \nabla_t(y_x) \text{ for some } y_x \in V_{un}\}. \end{aligned}$$

Moreover, there are two injections:  $h_\nu : V_w^\nu \rightarrow fVar(t)$  and  $h_{un} : V_w^{un} \rightarrow V_{un}$  given by  $h_\nu(x) = \nu_x$  and  $h_{un}(x) = y_x$ . Let us write  $V_r^\nu(t) = fVar(t) \setminus h_\nu(V_w^\nu)$ ,  $V_r^{un} = V_{un} \setminus h_{un}(V_w^{un})$ ,  $V_w = V_w^\nu \cup V_w^{un}$ ,  $V_b = V_d \cup V_{un}$  and  $V_r = V_r^\nu \cup V_r^{un}$ .

For all  $x \in V_{un}$ ,  $\sigma(x)$  is a product in the unbounded part. For all  $x \in V_d$ , the products  $\sigma(x)$  are mapped to themselves by  $\iota$ , so that  $\nabla_x$  will be used instead in the following firing of  $t$ . They will be responsible for d-accelerations. Products  $\sigma(x)$  with  $x \in V_w^\nu$  are those mapped by  $\iota$  to fresh products. Therefore,  $post_t(\nu_x)$  will be used instead in the next firing, so that it will leave some garbage that will cause a w-acceleration. Other products of the form  $post_t(\nu)$  will not be used later, those with  $\nu \in V_r^\nu$ , so that they will also contribute to the w-acceleration.

Variables in  $V_w^{un}$  and  $V_r^{un}$  have an effect analogous to those in  $V_w^\nu$  and  $V_r^\nu$ . Products  $\sigma(x)$  with  $x \in V_w^{un}$  are mapped by  $\iota$  to a product  $\nabla_{y_x}$  that has evolved from a product in the unbounded part. As before,  $\nabla_{y_x}$  will be used instead in the next firing, leaving again some garbage. Moreover, some products  $\nabla_y$  that come from a product in the unbounded part (those with  $y \in V_r^{un}$ ) will also remain and contribute to the w-acceleration.

**Definition 10.** Let  $\mathcal{M}_1 \xrightarrow{t(\sigma)} \mathcal{M}_2$  be a properly increasing sequence. We define the following products:

- For all  $x \in V_d$ ,  $\Delta_t(x)$  is any product such that  $\sigma(x) \xrightarrow{t(\sigma)} \sigma(x) \oplus \Delta_t(x)$ ,
- For all  $x \in V_w^\nu$ ,  $\Delta_t^t(x) = post_t(h_\nu(x)) \ominus pre_t(x)$ ,
- For all  $x \in V_w^{un}$ ,  $\Delta_t^t(x) = \nabla_t(h_{un}(x)) \ominus \sigma(x)$ .

<pre> <b>Procedure</b> Clover(<math>M_0</math>) <math>\Theta \leftarrow \{M_0\}</math> <b>while</b> <math>\overline{Post}(\Theta) \not\sqsubseteq \Theta</math> <b>do</b>   Choose fairly <math>\mathcal{M} \in \Theta</math>, <math>\tau</math> and <math>\iota</math>   such that <math>\mathcal{M} \xrightarrow{\tau} \mathcal{M}'</math>   <b>if</b> <math>\mathcal{M} \not\sqsubseteq_{\iota} \mathcal{M}'</math> <b>then</b>     <math>\Theta \leftarrow \Theta \cup \{\mathcal{M}'\}</math>   <b>else</b>     <math>\Theta \leftarrow \Theta \cup \{acc_{\iota}(\mathcal{M} \xrightarrow{\tau} \mathcal{M}')\}</math> <b>return</b> <math>Max \ \Theta</math>                 </pre>	<pre> <b>Procedure</b> width-Clover(<math>M_0</math>) <math>\Theta \leftarrow \{M_0\}</math>, <math>bounded \leftarrow \mathbf{true}</math> <b>while</b> <math>\overline{Post}(\Theta) \not\sqsubseteq \Theta</math> <b>and</b> <math>bounded</math> <b>do</b>   Choose fairly <math>\mathcal{M} \in \Theta</math>, <math>\tau</math> and <math>\iota</math> such that <math>\mathcal{M} \xrightarrow{\tau} \mathcal{M}'</math>   <b>if</b> <math>\mathcal{M} \not\sqsubseteq_{\iota} \mathcal{M}'</math> <b>then</b>     <math>\Theta \leftarrow \Theta \cup \{\mathcal{M}'\}</math>   <b>else</b>     <math>\mathcal{M}' \leftarrow acc_{\iota}(\mathcal{M} \xrightarrow{\tau} \mathcal{M}')</math>     <b>if</b> <math>x</math>-bounded(<math>\mathcal{M}'</math>) <b>then</b>       <math>\Theta \leftarrow \Theta \cup \{\mathcal{M}'\}</math>     <b>else</b>       <math>bounded \leftarrow \mathbf{false}</math> <b>return</b> (<math>bounded, Max \ \Theta</math>)                 </pre>
---	--

**Fig. 7.** Karp-Miller procedure (left) and algorithm deciding width-boundedness (right)

**Proposition 3.** *If  $\mathcal{M}_1 \xrightarrow{t(\sigma)} \mathcal{M}_2$  is properly increasing with  $\mathcal{M}_1 \sqsubseteq_{\iota} \mathcal{M}_2$  then there is  $\mathcal{M}$  such that  $\mathcal{M}_1 \equiv \sum_{x \in V_b} \sigma(x) + \infty(\sum_{x \in V_{un}} \sigma(x)) + \mathcal{M}$ , and  $acc_{\iota}(\mathcal{M}_1 \xrightarrow{t(\sigma)} \mathcal{M}_2)$  is*

$$\sum_{x \in V_d} (\sigma(x) \oplus \Delta_t(x)^{\omega}) + \sum_{x \in V_w^v} (post_t(h_{\nu}(x))) + \sum_{x \in V_w^{un}} (\sigma(x) \oplus \Delta_t^{\iota}(x)) + \sum_{x \in V_w} \nabla_t(x) + \infty(\sum_{x \in V_w^v} (post_t(x) \oplus \Delta_t^{\iota}(x))) + \sum_{x \in V_w^{un}} (\nabla_t(x) \oplus \Delta_t^{\iota}(x)) + \sum_{x \in V_r} \nabla_t(x) + \sum_{x \in V_{un}} \sigma(x) + \mathcal{M}$$

Moreover, the computation of the acceleration does not depend on the increment  $\Delta_t(x)$  chosen.

**Corollary 2.** *The completion of a  $\nu$ -APN is an  $\infty$ -effective (complete) WSTS.*

Because it is  $\infty$ -effective, it makes sense to apply the Karp-Miller procedure  $Clover(M_0)$  in the left of Fig. 7. Fairness in the choosing of the tuples  $(\mathcal{M}, \tau, \iota)$  ensures that in every infinite run, every such tuple will be eventually chosen at a later stage. We know that the cover is effectively representable, so that there is a finite set of  $\omega$ -markings  $\Theta$  such that  $\downarrow Post^*(\downarrow M_0) = \bigcup_{\mathcal{M} \in \Theta} \llbracket \mathcal{M} \rrbracket$ .

*Example 1.* Let us see with detail how the algorithm behaves for the  $\nu$ -APN  $N_2$  in Fig. 5. The initial  $\omega$ -marking is  $\mathcal{M}_0 = p + q$ , that is,  $\Theta = \{\mathcal{M}_0\}$ . The only possible mode that enables  $t$  is given by  $\sigma_1(x) = p$  and  $\sigma(y) = q$ , which produces the  $\omega$ -marking  $p + qq$ . Notice that:

- $p + q \sqsubseteq_{\iota_1} p + qq$ , with  $\iota_1(p) = p$  and  $\iota_1(q) = qq$ ,
- the product  $p$  in  $\mathcal{M}_0$  disappears,
- the product  $q$  in  $\mathcal{M}_0$  evolves to  $p$ ,
- the product  $qq$  in  $p + qq$  is fresh.

Therefore, the firing is not properly increasing, because there is a product  $p$  in  $p + qq$ , and two different products in  $\mathcal{M}_0$ , namely  $p$  and  $q$ , such that  $p$  is mapped



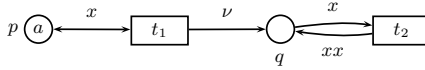


Fig. 8. dw-accelerations

to  $p$  by  $\iota_1$  and  $q$  evolves to  $p$ . Actually, we are exactly in the situation of the diagram in the left of Fig. 3. Therefore, we need to unroll the transition sequence  $t(\sigma_1)$ , with contraction depicted in Fig. 6. There, the firing  $p + q \rightarrow pq + qq$  can take place, which is properly increasing. Moreover,  $V_d = V_{un} = V_w^{un} = \emptyset$  and  $V_w^\nu = \{x, y\}$  with  $h_\nu(x) = \nu_1$  and  $h_\nu(y) = \nu_2$ .

- $\Delta_t(x) = post_t(\nu_1) \ominus pre_t(x) = q,$
- $\Delta_t(y) = post_t(\nu_2) \ominus pre_t(y) = q,$
- $\nabla_t(x) = (\sigma(x) \ominus pre_t(x)) \oplus post_t(x) = \emptyset,$
- $\nabla_t(y) = (\sigma(y) \ominus pre_t(y)) \oplus post_t(y) = \emptyset.$

Then, according to Prop. 3, the acceleration is  $pq + qq + \infty(q + q) \equiv pq + qq + \infty(q) = \mathcal{M}_1$ , so that  $\Theta = \{\mathcal{M}_0, \mathcal{M}_1\}$ . Starting from  $\mathcal{M}_1$  we could fire  $t(\sigma)$  with  $\sigma(x) = pq$  and  $\sigma(y) = qq$ , that produces again the  $\omega$ -marking  $\mathcal{M}_1$ . We can also fire  $t$  from  $\mathcal{M}_1$  with a different mode  $\sigma$ , with  $\sigma(x) = pq$  and  $\sigma(y) = q$ , which yields the  $\omega$ -marking  $\mathcal{M}_2 = p + qq + qq + \infty(q)$ . Since  $\mathcal{M}_1 \not\sqsubseteq \mathcal{M}_2$  no acceleration is performed, and  $\Theta = \{\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2\}$ .

Let us now see what happens if we fire the transition starting from  $\mathcal{M}_2$ . We could fire it using a mode such that  $\sigma(x) = p$  and  $\sigma(y) = qq$ . The corresponding firing is increasing, but not properly increasing. As happened before, the order of the mapping  $\iota$  is 2, and the contraction of the unrolling is given again by Fig. 6. The acceleration is analogous to the one obtained from  $\mathcal{M}_0$ , and produces again the  $\omega$ -marking  $\mathcal{M}_1$ .

The other way in which  $t$  can be fired from  $\mathcal{M}_2$  is more interesting, namely in a mode  $\sigma$  with  $\sigma(x) = p$  and  $\sigma(y) = q$ . Notice that  $y$  is instantiated to a product in the unbounded part of  $\mathcal{M}_2$ . Using that mode, the firing  $p + qq + \infty(q) \rightarrow p + qq + qq + \infty(q)$  can happen. Moreover, that firing is properly increasing. Indeed,  $\iota(p) = p$ ,  $\iota(qq) = qq$  (for both occurrences of  $qq$ ) and  $\iota(q) = q$  and, although the product  $\iota(p) = p$  is the result of the evolution of a product different from  $p$ , namely  $q$ , that product is in the unbounded part. Now we have  $V_d = V_w^\nu = V_r^{un} = \emptyset$ ,  $V_{un} = \{y\}$ ,  $V_w^{un} = \{x\}$  and  $V_r^\nu = \{\nu\}$ , with  $h_{un}(x) = y$ . Moreover,  $\Delta_t^i(x) = \nabla_t(x) = \emptyset$ , so that the acceleration is  $(p \oplus \emptyset) + \emptyset + \infty(\emptyset \oplus \emptyset + qq + q) + qq + qq \equiv p + \infty(qq) = \mathcal{M}_3$ .

Similarly, from  $\mathcal{M}_3$  we can obtain the  $\omega$ -marking  $\mathcal{M}_4 = pq + \infty(qq)$ , and obtain  $\Theta = \{\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4\}$ . From  $\Theta$ , no other  $\omega$ -marking can be obtained. Thus, the algorithm returns the maximal  $\omega$ -marking  $\mathcal{M}_4$  (because  $\mathcal{M}_i \sqsubseteq \mathcal{M}_4$  for all  $i$ ), so that the cover is the set of markings  $\llbracket pq + \infty(qq) \rrbracket$ . In particular, every reachable marking  $M$  has one identifier  $a$  and a (possibly empty) set of identifiers  $\{b_1, \dots, b_m\}$  such that  $M^a \subseteq \{p, q\}$  and  $M^{b_i} \subseteq \{q, q\}$ .

It is easy to see that the procedure  $\text{Clover}(M_0)$  does not terminate in general. Consider the net in Fig. 8. First,  $p \xrightarrow{t_1} p + q$  and we can apply a w-acceleration as previously explained, thus obtaining  $p + \infty(q)$ . Now we can fire transition  $t_2$ ,

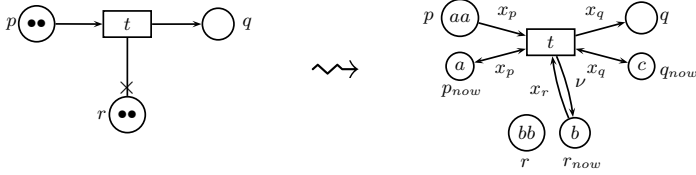


Fig. 9. Simulation of reset nets

$p + \infty(q) \xrightarrow{t_2} p + qq + \infty(q)$ . Notice that  $p + \infty(q) \sqsubseteq_{\iota} p + qq + \infty(q)$  with  $\iota(p) = p$  and  $\iota(q) = q$ . The algorithm could then replace  $p + qq + \infty(q)$  by its acceleration  $p + \infty(qq)$ . In the same way, all the  $\omega$ -markings  $p + \infty(q^n)$  are produced by the algorithm.

We could consider yet another type of acceleration, that we could call *dw-acceleration*. Instead of firing  $t_2$  again using one of the infinitely many  $q$ 's, we could fire it using  $qq$ . If we repeat this process, every marking  $p + \sum_{i=1}^m q^i + \infty(q)$  becomes reachable, and their least upper bound is  $p + \infty(q^w)$ .

It is true that dw-accelerations give a better approximation of the clover. However, they are not enough, neither any other acceleration we could imagine, since, in general, it is not possible to compute the clover.

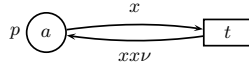
**Proposition 4.** *There is a  $\nu$ -APN for which the clover is not computable.*

*Proof (sketch).* Let us suppose that the clover is always computable. Let us see that, in that case, we could decide boundedness of reset nets, which we know to be undecidable [7]. Given a reset net  $N$  we build a  $\nu$ -APN  $N^*$  that simulates it. For each place  $p$  of  $N$  we consider a new place  $p_{now}$  in  $N^*$ . The construction of  $N^*$  is such that  $p_{now}$  contains a single token at any time. The firing of any transition ensures (by matching) that the token being used in  $p$  coincides with that in  $p_{now}$ . Every time a transition resets a place  $p$ , the content of  $p_{now}$  is replaced by a fresh token, so that no token remaining in  $p$  can be used. In this way, our simulation introduces some garbage tokens, that once become garbage, always stay like that. Fig. 9 depicts a simple reset net and its simulation.

Moreover, if  $M_0$  is the initial marking of  $N$ , we consider a different identifier  $a_p$  for each place  $p$  of  $N$ . Then, we define the initial marking of  $N^*$  as  $M_0^*(p_{now}) = \{a_p\}$  and  $M_0^*(p) = \{a_p, M_0(p), a_p\}$ , for each  $p \in P$ . Let us suppose that we can compute the clover  $\Theta$ . Then  $N$  is unbounded iff there is  $\mathcal{M} \in \Theta$  containing a product  $p_{now}p^\omega$ , for some  $p \in P$ , and boundedness would turn decidable.  $\square$

In particular, since Petri Data Nets [16] subsume  $\nu$ -APN, there is no procedure computing the cover of a Petri Data Net, neither for a Transfer Data Net, thus answering negatively to a question posed in [11].

**Accelerations and non-determinism.** In Def. 8 we are fixing by means of the mapping  $\iota$  the relation between names in  $\mathcal{M}_1$  and names in  $\mathcal{M}_2$ . In particular, we



**Fig. 10.** Accelerations and non-determinism

are choosing among one of such possible relations, and forcing that the chosen relation is kept between all the markings in the generated increasing chain. Thus, we are removing part of the inherent non-determinism in  $\nu$ -APN that arises in the non-deterministic choosing of consumed names by transitions. For instance, the net in Fig. 10 can fire its only transition  $p \xrightarrow{t} p + pp$  and  $p \sqsubseteq p + pp$ , but we can choose two different ways to map products to products, namely  $\iota_1(p) = p$  and  $\iota_2(p) = pp$ . In the first case, the result of accelerating is the  $\omega$ -marking  $\mathcal{M}_1 = \infty(pp)$  (we are always consuming the just created name), while in the second case we obtain  $\mathcal{M}_2 = p^\omega + \infty(p)$  (we are always taking the name that appeared already in the initial marking).

If we do not impose any particular relation between the names, then at any point any token could be chosen, so that starting from the initial marking, any marking of the form  $p^{n_1} + \dots + p^{n_k}$  can be reached, with least upper bound  $\infty(p^\omega)$ . Therefore, any acceleration schema that does not impose any mapping  $\iota$  relating names should compute  $\infty(p^\omega)$  as acceleration.

In general, if we can choose between several mappings  $\iota_1, \dots, \iota_k$ , because of monotonicity, in each of the limits  $\mathcal{M}_1, \dots, \mathcal{M}_k$  we can again choose between those mappings. Actually, if we choose again the mapping  $\iota_i$  starting from  $\mathcal{M}_i$ , the obtained marking is again  $\mathcal{M}_i$ , by definition of acceleration. However, we could use a different  $\iota_j$  to accelerate starting from  $\mathcal{M}_i$ , with  $i \neq j$ . In our previous example, we can again accelerate starting from  $\mathcal{M}_1 = \infty(pp)$  and from  $\mathcal{M}_2 = p^\omega + \infty(p)$ . In the case of  $\mathcal{M}_1$ , we reach  $\mathcal{M}_3 = ppp + \infty(pp)$ , and  $ppp$  is obtained from one of the infinitely-many  $pp$ . If we apply a dw-acceleration we obtain  $\infty(p^\omega)$ . Moreover, this is also what we obtain if we accelerate starting from  $\mathcal{M}_2$ . Indeed, we could choose to fire  $t$  starting from  $\mathcal{M}_2$  in a mode  $\sigma_1(x) = p^\omega$ , but the reached marking would be again  $\mathcal{M}_2$ . However, if we consider a mode  $\sigma_2(x) = p$ , then we reach  $p^\omega + pp + \infty(p)$ , where  $pp$  is obtained from one of the  $p$ . Thus, we can again apply a dw-acceleration to obtain the same  $\omega$ -marking  $\mathcal{M}_2$ .

In the previous example we have managed to accelerate (using dw-accelerations) without restricting ourselves to a given mapping  $\iota$ . However, it remains to see that we can do it in general, that is, that we can still accelerate any loop even if we remove the hypothesis of accelerating with respect to a given mapping  $\iota$ .

**Width-boundedness.** We have proved that the cover is not computable in general. To conclude the section we will use the generic Karp-Miller procedure (or a slight variation) to decide a property related to boundedness, called width-boundedness in [18].

**Definition 11.** We say  $N$  is width-bounded if there is  $n \in \mathbb{N}$  such that for all reachable  $M$ ,  $|Id(M)| \leq n$ .

Let us see that the forward analysis, though non-terminating in general, can decide width-boundedness. Let us define the predicate over  $\omega$ -markings `width-bounded`, given by `width-bounded`( $\mathcal{M}$ ) iff  $\mathcal{M} = \mathcal{A} + \infty(\emptyset)$ . To detect width-boundedness it is enough to stop whenever an  $\omega$ -marking  $\mathcal{M}$  such that  $\neg$ `width-bounded`( $\mathcal{M}$ ) is found. In this way we can slightly modify the procedure `Clover`, obtaining the algorithm in the right of Fig. 7, `width-Clover`( $M_0$ ). The modified algorithm always terminates, returning true iff the net is width-bounded, in which case the clover is computed.

**Proposition 5.** *Width-boundedness is decidable and the clover is computable for width-bounded  $\nu$ -APN.*

## 6 Conclusions and Future Work

In this paper we have established a forward analysis for  $\nu$ -APNs, an extension of P/T nets with pure name management and creation, with the goal of computing a finite basis of its cover, that is, of the set  $\downarrow Post^*(M_0)$ . For that purpose, we have applied the results and techniques developed in [10,11] for WSTS. We have defined a friendly presentation of the completion of a  $\nu$ -APN by means of  $\omega$ -markings, a natural extension of the analogous concept for P/T nets. We have seen that the transition relation, lifted to the completion, is effective (we can compute successors) and  $\infty$ -effective (we can compute the least upper bounds of the sets of  $\omega$ -markings produced by simple loops). This ensures that it makes sense to apply a forward Karp-Miller procedure. Unfortunately, we have proved that such procedure cannot terminate in general, or we could decide boundedness in reset nets, which is undecidable. As a corollary, a finite basis of the cover is not computable for the class of Transfer Data Nets, not even for the class of Petri Data Nets. Nevertheless, we can slightly modify that algorithm to get a procedure to decide width-boundedness and to compute a finite basis of the cover of a width-bounded net.

The d-accelerations and w-accelerations in Sect. 5 appear naturally when computing the least upper bound of simple loops. However, the dw-accelerations have been sketched in a rather ad-hoc way. It would be interesting to formalize the type of loops they accelerate, and possibly to characterize a subclass of  $\nu$ -APNs (larger than width-bounded nets) for which `Clover` terminates. In general, it would be interesting to see if a non-deterministic version of accelerations, in which we do not restrict the modes by the relation between the different names involved represented by the mapping  $\iota$ , is computable. More precisely, it would be interesting to study the structure of the set of markings  $\{\mathcal{M} \mid \mathcal{M}_1 \xrightarrow{\iota^k} \mathcal{M}\}$  (without restricting the modes), to see if it is a directed set, and computing its least upper bound in that case.

**Acknowledgments.** The authors would like to thank the anonymous referees for their valuable comments.

## References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: Algorithmic Analysis of Programs with Well Quasi-ordered Domains. *Inf. and Comp.* 160(1-2), 109–127 (2000)
2. Abdulla, P.A., Nylén, A.: Better is Better than Well: On Efficient Verification of Infinite-State Systems. In: 15th Annual IEEE Symp. on Logic in Computer Science, LICS 2000, pp. 132–140 (2000)
3. Abdulla, P.A., Collomb-Annichini, A., Bouajjani, A., Jonsson, B.: Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *Formal Methods in System Design* 25(1), 39–65 (2004)
4. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat Acceleration in Symbolic Model Checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005)
5. Dietze, R., Kudlek, M., Kummer, O.: Decidability Problems of a Basic Class of Object Nets. In: *Fundamenta Informaticae*, vol. 79, pp. 295–302. IOS Press, Amsterdam (2007)
6. Decker, G., Weske, M.: Instance Isolation Analysis for Service-Oriented Architectures. In: *Int. Conference on Services Computing, SCC 2008*, pp. 249–256. IEEE Computer Society, Los Alamitos (2008)
7. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset Nets Between Decidability and Undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)
8. Finkel, A., Schnoebelen, P.: Fundamental Structures in Well-Structured Infinite Transition Systems. In: Lucchesi, C.L., Moura, A.V. (eds.) LATIN 1998. LNCS, vol. 1380, pp. 102–118. Springer, Heidelberg (1998)
9. Finkel, A., Schnoebelen, P.: Well-Structured Transition Systems Everywhere! *Theoretical Computer Science* 256(1-2), 63–92 (2001)
10. Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, Part I: Completions. In: *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science, STACS'09*, pp. 433–444 (2009)
11. Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, Part II: Complete WSTS. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikolettseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 188–199. Springer, Heidelberg (2009)
12. Geeraerts, G., Raskin, J.-F., van Begin, L.: Expand, enlarge and check: New algorithms for the coverability problem of WSTS. *J. Comp. Sys. Sci.* 72(1), 180–203 (2006)
13. Gordon, A.: Notes on Nominal Calculi for Security and Mobility. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 262–330. Springer, Heidelberg (2001)
14. Goubault-Larrecq, J.: On Noetherian spaces. In: 22nd IEEE Symposium on Logic in Computer Science, LICS 2007, pp. 453–462. IEEE Computer Society, Los Alamitos (2007)
15. Jančar, P.: A note on well quasi-orderings for powersets. *Information Processing Letters* 72(5-6), 155–160 (1999)
16. Lazic, R., Newcomb, T., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with Tokens which Carry Data. *Fundamenta Informaticae* 88(3), 251–274 (2008)

17. Rosa-Velardo, F., de Frutos-Escrig, D., Marroquín-Alonso, O.: On the expressiveness of Mobile Synchronizing Petri Nets. In: 3rd International Workshop on Security Issues in Concurrency, SecCo 2005. ENTCS, vol. 180(1), pp. 77–94. Elsevier, Amsterdam (2007)
18. Rosa-Velardo, F., de Frutos-Escrig, D.: Name creation vs. replication in Petri Net systems. *Fundamenta Informaticae* 88(3), 329–356 (2008)
19. Rosa-Velardo, F., de Frutos-Escrig, D.: Name creation vs. replication in Petri Net systems. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 402–422. Springer, Heidelberg (2007)
20. Schnoebelen, P.: Verifying lossy channel systems has nonprimitive recursive complexity. *Inf. Process. Lett.* 83(5), 251–261 (2002)

# Learning Workflow Petri Nets

Javier Esparza, Martin Leucker, and Maximilian Schlund

Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany  
{esparza,leucker,schlund}@in.tum.de

**Abstract.** Workflow mining is the task of automatically producing a workflow model from a set of event logs recording sequences of workflow events; each sequence corresponds to a use case or workflow instance. Formal approaches to workflow mining assume that the event log is complete (contains enough information to infer the workflow) which is often not the case. We present a learning approach that relaxes this assumption: if the event log is incomplete, our learning algorithm automatically derives queries about the executability of some event sequences. If a teacher answers these queries, the algorithm is guaranteed to terminate with a correct model. We provide matching upper and lower bounds on the number of queries required by the algorithm, and report on the application of an implementation to some examples.

## 1 Introduction

Modern workflow management systems offer modelling capabilities to support business processes [vdAvH04]. However, constructing a formal or semi-formal workflow model of an existing business process is a non-trivial task, and for this reason *workflow mining* has been extensively studied (see [vdAvDH<sup>+</sup>03] for a survey). In this approach, information about the business processes is gathered in form of logs recording sequences of workflow events, where each sequence corresponds to a use case. The logs are then used to extract a formal model. Workflow mining techniques have been implemented in several systems, most prominently in the ProM tool [vdAvDG<sup>+</sup>07], and successfully applied.

Most approaches to process mining use a combination of heuristics and formal techniques, like machine learning or neural networks, and do not offer any kind of guarantee about the relationship between the business process and the mined model. Formal approaches have been studied using *workflow graphs* [AGL98] and *workflow nets* [vdA98, BDLM07] as formalisms. These approaches assume that the logs provide enough information to infer the model, i.e., that there is one single model compatible with them. In this case we call the logs *complete*. This is a strong assumption, which often fails to hold, for two reasons: first, the number of use cases may grow exponentially in the number of tasks of the process, and so may the size of a complete set of logs. Second, many processes have “corner cases”: unusual process instances that rarely happen. A complete set of logs must contain at least one instance of each corner case.

In this paper we propose a learning technique to relax the completeness assumption on the set of logs. In this approach the model is produced by a *Learner*

that may ask questions to a *Teacher*. The Learner can have initial knowledge in the form of an initial set of logs; if the log contains enough information to infer the model, the Learner produces it. If not, it iteratively produces *membership queries* of the form: Does the business process have an instance (a use case) starting with a given sequence of tasks? For instance, in the standard example of complaint processing (see Figure 1 and [vdA98]), a membership query could have the form “Is there a use case in which first the complaint is registered and then immediately rejected?” The Teacher would answer no, because a decision on acceptance or rejection is made only after the customer has been sent a questionnaire.

Notice that the Learner does not guess the queries, they are automatically constructed by the learning algorithm. Under the assumption that the Teacher provides correct answers, the learning process is guaranteed to terminate with a correct model: a model whose executions coincide with the possible event sequences of the business process. In other words, we provide a formal framework with a correctness and completeness guarantee which only assumes the existence of the Teacher.

It could be objected that if a Teacher exists, then a workflow model must already exist, and there is no need to produce it. To see the flaw in this argument, observe that Teachers can be employees, databases of client records, etc., that have knowledge about the process, but usually lack the modelling expertise required to produce a formal model. Our learning algorithm only requires from the Teacher the low-level ability to recognize a given sequence of process actions as the initial sequence of process actions of some use case.

It is useful to draw an analogy. Witnesses of a crime can usually answer questions about the physical appearance of the criminal, but they are very rarely able to draw the criminal’s portrait: this requires interaction with a police expert. This interaction can be seen as a learning process: the Teacher is the witness, and the Learner is the police expert. The teacher has knowledge about the criminal, but is unable to express it in the form of a portrait. The Learner has the expertise required to produce a portrait, but needs input from the Teacher.

Like [vdA98, KRS06, BDLM07, RGvdA<sup>+</sup>07], we use *workflow nets*, introduced by van der Aalst, as formal model of business processes. Loosely speaking, a workflow net is a Petri net with a distinguished initial and final marking. Van der Aalst convincingly argues that well-formed business processes (an informal notion) correspond to *sound* workflow nets (a formal concept). A workflow net is *sound* [vdA98] if it is live and bounded. In this paper we follow van der Aalst’s ideas. Given a Teacher, we wish to learn a sound workflow net for the business process. It is easy to come up with a naive correct learning algorithm. However, a first naive complexity analysis yields that the number of queries necessary to learn a workflow net can be triple exponential in the number of tasks of the business process in the worst case. This seems to indicate that the approach is useless. However, we show how the special properties of sound workflow nets, together with a finer complexity analysis, lead to WNL, a new learning algorithm requiring a single exponential number of queries in the worst case. We also provide



an exponential lower bound, showing that WNL is asymptotically optimal. Finally, in a number of experiments we show that despite the exponential worst-case complexity the algorithm is able to synthesize interesting workflows. Notice also that the complexity is analysed for the case in which no initial event log is provided, that is, the case in which all knowledge has to be extracted from the Teacher by asking membership queries.

Technically, the triple exponential complexity of the naive algorithm is a consequence of the following three facts:

- (a) the size of a deterministic finite automaton (DFA) recognizing the language of a net with  $n$  transitions can be a priori double exponential in  $n$ ;
- (b) learning such a DFA using only membership queries requires exponentially many queries in the size of the DFA (follows from [Ang87] and [Vas73, Cho78]); and
- (c) the algorithms of Darondeau et al. for synthesis of Petri nets from regular languages [BBD95] are exponential in the size of the DFA.

In the paper we solve (a) by proving that the size of the DFA is only single exponential; we solve (b) by exhibiting a better learning algorithm for sound workflow nets requiring only polynomially many queries; finally, we solve (c) by showing that for sound workflow nets the algorithms for synthesis of Petri nets from regular languages can be replaced by the algorithms for synthesis of bounded nets from minimal DFA, which are of polynomial complexity. Notice that our solution very much profits from the restriction to sound workflow nets, but that this restriction is given by the application domain: that sound workflow nets are an adequate formalization of well-formed business processes has been proved by the large success of the model in both the workflow modelling and Petri net communities.

*Outline.* In the next section, we fix the notation of automata, recall the notion of Petri nets and workflow nets, and cite results on synthesis of Petri nets from automata. Our learning algorithm WNL is elaborated in Section 3. Section 4 reports on our implementation and experimental results. Finally, we sum up our contribution in the conclusion.

## 2 Preliminaries

We assume that the reader is familiar with elementary notions of graphs, automata and net theory. In this section we fix some notations and define some less common notions.

**Automata and Languages.** A deterministic finite automaton (DFA) is a 5-tuple  $A = (Q, \Sigma, \delta, q_0, F)$  where  $Q$  is a finite set of *states*,  $\Sigma$  is a finite *alphabet*,  $q_0 \in Q$  is the *initial state*,  $\delta: Q \times \Sigma \rightarrow Q$  is the (partial) *transition function* and  $F \subseteq Q$  is the set of *final states*. We denote by  $\hat{\delta}$  the function  $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$  inductively defined by  $\hat{\delta}(q, \epsilon) = q$  and  $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ . The language

$\mathcal{L}(q)$  of some state  $q \in Q$  is the set of words  $w \in \Sigma^*$  such that  $\widehat{\delta}(q, w) \in F$ . The language recognized by a DFA  $A$  is defined as  $\mathcal{L}(A) := \mathcal{L}(q_0)$ . A language is *regular* if it is accepted by some DFA.

*Myhill-Nerode’s theorem and minimal DFAs.* Given a language  $L \subseteq \Sigma^*$ , we say two words  $w, w' \in \Sigma^*$  are *L-equivalent*, denoted by  $w \sim_L w'$ , if  $wv \in L \Leftrightarrow w'v \in L$  for every  $v \in \Sigma^*$ . The language  $L$  is regular iff *L-equivalence* partitions  $\Sigma^*$  into a finite number of equivalence classes. Given a regular language  $L$ , there exists a unique DFA  $A$  up to isomorphism with a minimal number of states such that  $\mathcal{L}(A) = L$ ; this automaton  $A$  is called the *minimal DFA* for  $L$ . The number of states of this automaton is equal to the number of equivalence classes.

Given a DFA  $A = (Q, \Sigma, \delta, q_0, F)$ , we say two states  $q, q' \in Q$  are *A-equivalent* if  $\mathcal{L}(q) = \mathcal{L}(q')$ . We can quotient  $A$  with respect to this equivalence relation. The states of the quotient DFA are the equivalence classes of  $\sim_A$ . The transitions are defined by “lifting” the transitions of  $A$ : for every transition  $q \xrightarrow{a} q'$ , add  $[q] \xrightarrow{a} [q']$  to the transitions of the quotient DFA, where  $[q]$  and  $[q']$  denote the equivalence classes of  $q$  and  $q'$ . The initial state is  $[q_0]$ , and the final states are  $\{[q] \mid q \in F\}$ . The quotient DFA recognizes the same language as  $A$ , and is isomorphic to the minimal DFA recognizing  $L$ .

It is easy to see that the minimal automaton for a prefix-closed regular language has a unique non-final state (a trap state). For simplicity, we sometimes identify this automaton with the one obtained by removing the trap state together with its ingoing and outgoing transitions.

**Petri Nets.** A (marked) Petri net is a 5-tuple  $N = (P, T, F, W, m_0)$  where  $P$  is a set of *places*,  $T$  is a set of *transitions* with  $P \cap T = \emptyset$ ,  $F \subseteq (P \times T) \cup (T \times P)$  is a *flow relation*,  $W : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$  is a *weight function* satisfying  $W(x, y) > 0$  iff  $(x, y) \in F$ , and  $m_0 : P \rightarrow \mathbb{N}$  is a mapping called the *initial marking*.

For each transition or place  $x$  we call the set  $\bullet x := \{y \in P \cup T : (y, x) \in F\}$  the *preset* of  $x$ . Analogously we call  $x^\bullet := \{y \in P \cup T : (x, y) \in F\}$  the *postset* of  $x$ . A net is *pure* if no transition belongs to both the pre- and postsets of some place.

Given an arbitrary but fixed numbering of  $P$  and  $T$ , the *incidence matrix* of  $N$  is the  $|P| \times |T|$ -matrix  $C$  given by:  $C(p_i, t_j) = W(t_j, p_i) - W(p_i, t_j)$ .

A transition  $t \in T$  is *enabled* at a marking  $m$ , if  $\forall p \in \bullet t : m(p) \geq W(p, t)$ . If a transition  $t$  is enabled it can *fire* to produce the new marking  $m'$ , written as  $m \xrightarrow{t} m'$ .

$$m'(p) := m(p) + \sum_{p' \in P} C(p', t)$$

Given  $w = t_1 \cdots t_n \in T^*$  (i.e.  $t_i \in T$ ), we write  $m_0 \xrightarrow{w} m$  if there exist markings  $m_1, \dots, m_{n-1}$  such that  $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \dots m_{n-1} \xrightarrow{t_n} m$ . Then, we say that  $m$  is *reachable*. The set of reachable markings of  $N$  is denoted by  $\mathcal{M}(N)$  and defined by  $\mathcal{M}(N) = \{m : \exists w \in T^*. m_0 \xrightarrow{w} m\}$ . It is well-known that if

$m_0 \xrightarrow{w} m$ , then  $m = m_0 + C \cdot P(w)$ , where  $P(w)$ , the *Parikh vector* of  $w$ , is the vector of dimension  $|T|$  having as  $i$ -th component the number of times that  $t_i$  occurs in  $w$ . We call this equality the *marking equation*.

A net  $N$  is *k-bounded* if  $m(p) \leq k$  for every reachable marking  $m$  and every place  $p$  of  $N$ , and *bounded* if it is  $k$ -bounded for some  $k \geq 0$ . A 1-bounded net is also called *safe*. A net is *reversible* if for every firing sequence  $m_0 \xrightarrow{w} m$  there is a sequence  $v_w$  leading back to the initial state, i.e.  $m \xrightarrow{v_w} m_0$ .  $N$  is *live* if every transition can fire eventually at every marking, i.e.  $\forall m \in \mathcal{M}(N) \exists w_m. m \xrightarrow{w_m t} m'$  for some  $m'$ .

The *reachability graph* of a net  $N = (P, T, F, W, m_0)$  is the directed graph  $G = (V, E)$  with  $V = \mathcal{M}(N)$  and  $(x, y) \in E$  iff  $x \xrightarrow{t} y$  for some  $t \in T$ . If  $G$  is finite, then the five-tuple  $A(N) = (Q, \Sigma, \delta, q_0, F)$  given by  $Q = \mathcal{M}(N)$ ,  $\Sigma = T$ ,  $q_0 = m_0$ ,  $F = Q$  and  $\delta(m, t) := m'$  if  $m \xrightarrow{t} m'$  is a DFA, and undefined otherwise. (Note that  $\delta$  is well-defined, because if  $m \xrightarrow{t} m'$  and  $m \xrightarrow{t} m''$  then  $m' = m''$ .) We call it the *marking-DFA* of  $N$ . The *language* of  $N$ , denoted by  $\mathcal{L}(N)$ , is defined as the language of  $A(N)$ .

**Workflow nets.** Loosely speaking, a workflow net is a Petri net with two distinguished input and output places without input and output transitions respectively, and such that the addition of a “reset” transition leading back from the output to the input place makes the net strongly connected (see Figure 11 for example). Formally, a net  $N = (P, T, F, W, m_0)$  is a *workflow net* if there exist places  $i, o \in P$  such that  $\bullet i = \emptyset = o \bullet$ ,  $m_0(p) = 1$  for  $p = i$  and  $m_0(p) = 0$ , otherwise, and the net  $\tilde{N} = (P, T \cup \{\mathbf{r}\}, F \cup \{(o, \mathbf{r}), (\mathbf{r}, i)\}, W \cup \{(o, \mathbf{r}) \mapsto 1, (\mathbf{r}, i) \mapsto 1\}, m_0)$ , where  $\mathbf{r} \notin T$ , is strongly connected.

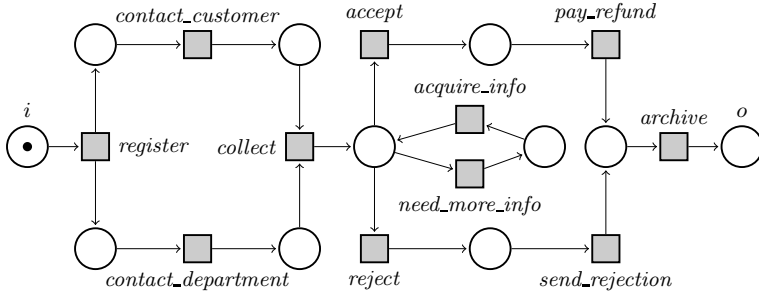
A firing sequence  $w$  of a workflow net  $N$  is a *run* if  $m_0 \xrightarrow{w\mathbf{r}} m_0$  in  $\tilde{N}$ . The runs of  $N$  are the formalization of the use cases of the business process modelled by the workflow net. A workflow net  $N$  is *sound* if  $\tilde{N}$  is live and bounded. It is argued in [vdA98] that a well-formed business process can be modelled by a sound workflow net (at a certain level of abstraction). The workflow net in Figure 11 is a very simple model for processing complaints (a slightly altered example, taken from [vdAvH04]).

The following lemma characterizes soundness. In the paper we work with this characterization as definition.

**Lemma 1.** *A workflow net  $N$  is sound iff  $\tilde{N}$  is bounded, reversible, and for every transition  $t$  there is a reachable marking  $m$  such that  $m$  enables  $t$ .*

*Proof.* Let  $N = (P, T, F, W, m_0)$  be workflow net.

( $\Rightarrow$ ): Assume  $N$  is sound. Then  $\tilde{N}$  is bounded and live. We show  $\tilde{N}$  is reversible. Let  $m$  be an arbitrary reachable marking of  $\tilde{N}$ . Then  $m_0 \xrightarrow{w} m$  for some  $w \in (T \cup \{\mathbf{r}\})^*$ . Since  $\tilde{N}$  is live, there is a firing sequence  $w$  such that  $m \xrightarrow{w\mathbf{r}} m'$  for some marking  $m'$ . We claim  $m' = m_0$ . Assume  $m' \neq m_0$ . Then, since  $m'(i) > 0$ , we have  $m'(p) \geq m_0(p)$  for every place  $p$ , and  $m'(p) > m_0(p)$  for some  $p$ . So  $m'$  strictly covers  $m_0$ , and so  $N$  is not bounded.



**Fig. 1.** An example for a sound workflow net (drawn without the reset transition  $\mathbf{r}$ )

( $\Leftarrow$ ): Assume  $\tilde{N}$  is bounded, reversible and every transition is enabled at some reachable marking. We show that  $\tilde{N}$  is live, which implies that  $N$  is sound. Let  $m$  be an arbitrary reachable marking of  $\tilde{N}$ , and let  $t \in T \cup \{\mathbf{r}\}$ . Since  $\tilde{N}$  is reversible,  $m \xrightarrow{w} m_0$  for some  $w \in (T \cup \{\mathbf{r}\})^*$ , and since  $t$  occurs in some firing sequence  $m_0 \xrightarrow{vt} m'$  for some  $v \in (T \cup \{\mathbf{r}\})^*$  and some  $m'$ . So  $\tilde{N}$  is live (and bounded by assumption) and therefore  $N$  is sound.

**Synthesis of Petri nets from Languages and from Automata.** In [BBD95], Darondeau et al. address two synthesis problems of Petri nets from a minimal DFA  $A$  over an alphabet  $T$ :

- (S1) Decide if there is a bounded net  $N$  with  $T$  as set of transitions such that  $\mathcal{L}(N) = \mathcal{L}(A)$ , and if so return one. We call this problem *synthesis up to language equivalence*.
- (S2) Decide if there is a bounded net  $N$  with  $T$  as set of transitions such that the reachability graph of  $N$  is isomorphic to  $A$ , and if so return one. We call this problem *synthesis up to isomorphism*.

The algorithm of [BBD95] for synthesis up to language equivalence works in two phases: firstly,  $A$  is transformed into an equivalent automaton  $A'$  in a certain normal form. In the worst case,  $A'$  can be exponentially larger than  $A$ . The second phase constructs the net  $N$ , if it exists, in polynomial time in  $A'$ . The algorithm requires *exponential time* in  $A$ . The algorithm of [BBD95] for synthesis up to isomorphism, on the contrary, needs only *polynomial time* in  $A$ . Notice that, in general, if one knows the language  $\mathcal{L}(N)$  of a net, one does not know directly its reachability graph. In particular, the minimal automaton recognizing  $\mathcal{L}(N)$  may not be the reachability graph of any net. The basic algorithm in [BBD95] can only handle pure nets, but there is also a generalization to non-pure nets to be found in [BDBM96].

Hints on how to obtain nets that are more “visually appealing” (i.e. have few arcs, no redundant places, etc.) than those generated by standard synthesis algorithms can be found in [BDKM08], where net synthesis was applied to process mining from event logs.

### 3 A Learning Algorithm for Sound Workflow Nets

Our goal is to develop a learning algorithm for sound workflow nets which is guaranteed to terminate, and in which a teacher only needs to answer membership queries.

The precise learning setting is as follows. We have a *Leaner* and a *Teacher*. The Learner is given a set  $T$  of transitions, where each transition corresponds to a dedicated *task* (in the sense of [vdA98]) of the business process. The Learner repeatedly asks the Teacher *workflow membership queries*. A query is a sequence  $\sigma \in T^*$ , and is answered by the Teacher as follows: if  $\sigma$  can be extended to a use case (i.e., a sequence corresponding to a complete instance of the business process), then the Teacher returns this use case in the form of a transition sequence  $\sigma\tau\mathbf{r}$ , where  $\tau \in T^*$ . Otherwise, the Teacher answers “no”. In our running example the Learner is given the set of transitions of the net of Figure 1, and the Teacher’s answers are compatible with this net, i.e., acts as if it knew the net. Note that in practice, this only means that the Teacher can either extend the query to a use case of the net to learn or can reject the query. Two possible queries are

*register contact\_customer contact\_department*  
*register contact\_customer collect*

A possible answer to the first query is the run

*register contact\_customer contact\_department collect accept pay\_refund archive*

while the answer to the second query is “no”.

Assuming that the Teacher’s answers are compatible with a  $k$ -bounded and reversible net  $N$ , the goal of the Learner is to produce a net  $N'$  such that  $\mathcal{L}(N) = \mathcal{L}(N')$ . It is easy to see that a (very inefficient) learning algorithm exists:

- (1) A net with  $n$  transitions has at most  $c_1 := 2^{(n+1)}$  places, because a place is determined by its pre- and post-sets of transitions.
- (2) By (1),  $N$  has at most  $c_2 := (k+1)^{c_1}$  reachable markings. Therefore, there exists a minimal DFA  $A$  with at most  $c_2$  states such that  $\mathcal{L}(N) = \mathcal{L}(A)$ .
- (3) Since any two prefix-closed minimal DFAs with  $c_2$  states differ in some word of length  $c_2$ , the automaton  $A$  can be learned by querying all words over  $T$  of length  $2c_2$ , i.e., after at most  $c_3 := n^{2c_2}$  queries.

This follows easily from Myhill-Nerode’s theorem. The DFA  $A$  can be constructed from the answers to the queries as follows. The states of  $A$  are the equivalence classes of words of  $\mathcal{L}(N)$  of length up to  $c_2$ , where two words  $w, v$  are equivalent if for every word  $u$  of length up to  $c_2$  either  $wu$  and  $vu$  belong to  $\mathcal{L}(N)$ , or none of them does [Vas73, Cho78]. The initial state is the equivalence class of the empty word, and all states are final. There is a transition  $[w] \xrightarrow{a} [wa]$  for every word  $w$  of length at most  $c_2$ .

- (4) The net  $N$  is obtained from  $A$  by means of the algorithm of [BBD95] for synthesis up to language equivalence (see problem (S1) in Section 2). The algorithm runs in  $2^{\mathcal{O}(p(c_2))}$  time for some polynomial  $p$ .

The query complexity of this naive algorithm, i.e. the number of queries it needs to ask, is triple exponential in the number  $n$  of transitions. In this section we prove a series of results ending in an improved algorithm with single exponential query and time complexity (notice that single exponential time complexity implies single exponential query complexity, but not vice versa).

### 3.1 An Upper Bound on the Number of Reachable Markings

We show that the naive bound on the number of states of  $A$  obtained in (2) above, which is double exponential in  $n$ , can be improved to a single exponential bound.

Given a net  $N = (P, T, F, W, m_0)$  with incidence matrix  $C$ , we denote by  $C(p)$  the vector  $(C(p, t_1), \dots, C(p, t_{|T|}))$ . We say that a place  $p$  is a *linear combination* of the places  $p_1, \dots, p_k$  if there are real numbers  $\lambda_1, \dots, \lambda_k$  such that  $C(p) = \sum_{i=1}^k \lambda_i \cdot C(p_i)$ .

The following lemma is well known.

**Lemma 2.** *Let  $N = (P, T, F, W, m_0)$  be a net with incidence matrix  $C$ , and let  $C(p) = \sum_{i=1}^k \lambda_i C(p_i)$ . Then for every reachable marking  $m$ :  $\forall p \in P. m(p) = m_0(p) + \sum_{i=1}^k \lambda_i (m(p_i) - m_0(p_i))$ .*

*Proof.* Since  $m$  is reachable, there is  $w \in T^*$  such that  $m_0 \xrightarrow{w} m$ . By the marking equation  $m = m_0 + C \cdot P(w)$ , and so in particular  $m(p) = m_0(p) + C(p) \cdot P(w)$ , and  $m(p_i) = m_0(p_i) + C(p_i) \cdot P(w)$  for every  $1 \leq i \leq k$ . So  $m(p) = m_0(p) + \sum_{i=1}^k \lambda_i C(p_i) \cdot P(w) = m_0(p) + \sum_{i=1}^k \lambda_i (m(p_i) - m_0(p_i))$

**Theorem 1.** *Let  $N = (P, T, F, W, m_0)$  be a  $k$ -bounded net with  $n$  transitions. Then  $N$  has at most  $(k+1)^n$  reachable markings.*

*Proof.* The incidence matrix  $C$  has  $|P|$  rows and  $n$  columns, and so it has rank at most  $n$ . So there are  $l$  places  $p_1, \dots, p_l$ ,  $l \leq n$ , such that  $C(p_1), \dots, C(p_l)$  are linearly independent. So every place  $p$  is a linear combination of  $p_1, \dots, p_l$ . It follows from Lemma 2 that for every two reachable markings  $m, m'$ , if  $m(p_i) = m'(p_i)$  for every  $1 \leq i \leq l$ , then  $m(p) = m'(p)$  for every place  $p$ . In other words, if two markings coincide on all of  $p_1, \dots, p_l$ , they are equal. Since for every reachable marking  $m$  we have  $0 \leq m(p_i) \leq k$ , the number of projections of the reachable markings onto the places  $p_1, \dots, p_l$  is at most  $(k+1)^l \leq (k+1)^n$ . So  $N$  has at most  $(k+1)^n$  reachable markings.

### 3.2 Minimality of the Marking-DFA

We show that the marking-DFA of a bounded and reversible net is minimal. Since our goal is to construct a bounded and reversible net model  $N$  of the business process, after we learn the minimal DFA  $A$  with  $\mathcal{L}(A) = \mathcal{L}(N)$  in step (3), we can synthesize  $N$  by applying the algorithm of [BBD95] for synthesis up to isomorphism (Problem (S2)), instead of the algorithm for synthesis up to

language equivalence (Problem (S1)). This eliminates one exponential from step (4) of the naive algorithm.

The proof is based on Lemma 3 below. Readers familiar with Myhill-Nerode's theorem (see also Section 2) will probably need no proof, but we include one for completeness. Recall that we identify a DFA with a single trap state with the one obtained by removing the trap state together with its ingoing and outgoing transitions.

**Lemma 3.** *A DFA  $A = (Q, \Sigma, \delta, q_0, F)$  is minimal iff the following two conditions hold:*

- (1) *every state lies in a path leading from  $q_0$  to some state of  $F$ , and*
- (2)  *$\mathcal{L}(q) \neq \mathcal{L}(q')$  for every two distinct states  $q, q' \in Q$ .*

*Proof.* ( $\Rightarrow$ ): We prove the contrapositive. For (1), if some state  $q$  does not lie in any path from  $q_0$  to some final state, then it can be removed without changing the language, and so  $A$  is not minimal. For (2), if two distinct states  $q, q'$  of  $A$  satisfy  $\mathcal{L}(q) = \mathcal{L}(q')$ , then  $[q] = [q']$ , and so the quotient automaton has fewer states than  $A$ . So  $A$  is not minimal.

( $\Leftarrow$ ): Assume (1) and (2) hold. We prove that for every state  $q$  the language of the words  $w$  such that  $\delta(q_0, w) = q$  is an equivalence class of  $L$ -equivalence. It follows that the number of states of  $A$  is at most as large as the number of equivalence classes of  $L$ -equivalence, which implies that  $A$  is the minimal DFA for  $L$ .

It suffices to show:

- If  $\widehat{\delta}(q_0, w) = q = \widehat{\delta}(q_0, v)$ , then  $w \sim_L v$ .

This follows immediately from the definition of  $L$ -equivalence.

- If  $\widehat{\delta}(q_0, w) = q$  and  $\widehat{\delta}(q_0, v) = q'$  for some  $q' \neq q$ , then  $w \not\sim_L v$ .

Since  $\mathcal{L}(q) \neq \mathcal{L}(q')$ , w.l.o.g. there is a word  $u \in \mathcal{L}(q) \setminus \mathcal{L}(q')$ . So  $wu \in L$  and  $vu \notin L$ , which implies  $w \not\sim_L v$ .

**Theorem 2.** *Let  $N = (P, T, F, W, m_0)$  be a bounded and reversible Petri net. The marking-DFA  $A(N)$  of  $N$  is a minimal DFA.*

*Proof.* Assume that  $A(N)$  is not minimal. Since every state of  $A(N)$  is final, by Lemma 3 there are two states of  $A(N)$ , i.e., two reachable markings  $m_1 \neq m_2$  of  $N$ , such that  $\mathcal{L}(m_1) = \mathcal{L}(m_2)$ . As  $m_1 \neq m_2$  there exists  $p \in P$  with  $m_1(p) \neq m_2(p)$ . Assume w.l.o.g.  $m_1(p) < m_2(p)$ . Let  $m$  be a reachable marking such that  $m(p)$  is minimal, i.e. there is no other reachable marking  $m'$  s.t.  $m'(p) < m(p)$ . Since  $m$  is reachable and  $N$  is reversible, there is  $w \in T^*$  such that  $m_2 \xrightarrow{w} m$ . Since  $\mathcal{L}(m_1) = \mathcal{L}(m_2)$ , there is a reachable marking  $m'$  such that  $m_1 \xrightarrow{w} m'$ . It follows

$$m'(p) = m_1(p) + C(p) \cdot P(w) < m_2(p) + C(p) \cdot P(w) = m(p)$$

contradicting the minimality of  $m(p)$ .

### 3.3 Learning the Reachability Graph by Exploration

The final step towards a single exponential learning algorithm consists of improving the naive algorithm in step (3) for learning the minimal DFA  $A$ . Recall that we assume that the Teacher's answers are compatible with a  $k$ -bounded and reversible net  $N$ . If  $n$  and  $r$  are the number of transitions and reachable markings of  $N$ , then the naive algorithm requires  $n^r$  membership queries. We present a new algorithm that requires only  $\mathcal{O}(n \cdot r^2)$  queries.

Recall the standard search approach for constructing the reachability graph of a net *if the net is known*. We maintain a queue of markings, initially containing the initial marking, and two sets of already visited markings and transitions (transitions between markings). While the queue is non-empty, we take the marking  $m$  at the top of the queue, and check for each transition  $a$  whether  $a$  is enabled at  $m$ . If so, we compute the marking  $m'$  such that  $m \xrightarrow{a} m'$ , and proceed as follows: if  $m'$  has been already visited, we add  $m \xrightarrow{a} m'$  to the set of visited transitions; if  $m'$  had not been visited yet, we add  $m'$  to the set of visited markings and to the queue, and add  $m \xrightarrow{a} m'$  to the set of visited transitions.

Our learning algorithm closely mimics this behaviour, but works with firing sequences of  $N$  instead of reachable markings (the Learner does not know the markings of the net, it does not even know its places). We maintain a queue of firing sequences, initially containing the empty sequence, and two sets of already visited firing sequences and transitions. While the queue is non-empty, we take the firing sequence  $w \in (T \cup \{\mathbf{r}\})^*$  at the top of the queue, and ask the Teacher for each transition  $a$  whether  $wa$  is also a firing sequence of  $N$ . If so, we proceed as follows. We first determine whether each already visited firing sequence  $u$  leads to the same marking as  $wa$ . Notice that it is not obvious how to do this—this is the key of the learning algorithm. If some firing sequence  $u$  leads to the same marking as  $wa$ , then we add  $w \xrightarrow{a} u$  to the set of visited transitions; otherwise, we add  $wa$  to the set of visited firing sequences and to the queue, and add  $w \xrightarrow{a} wa$  to the set of visited transitions. The algorithm in pseudo code can be found below (Algorithm [II](#)), where  $\text{Equiv}(u, v)$  denotes that there is a marking  $m$  such that  $m_0 \xrightarrow{u} m$  and  $m_0 \xrightarrow{v} m$ .

The correctness of the algorithm is immediate: we just simulate a search algorithm for the construction of the reachability graph, using a firing sequence  $u$  to represent the marking  $m$  such that  $m_0 \xrightarrow{u} m$ . The check  $\text{Equiv}(u, wa)$  guarantees that each marking gets exactly one representative.

The problem is to implement  $\text{Equiv}(u, wa)$  using only membership queries. In general this is no easy task, but in the case of reversible nets it can be easily done as follows. When checking  $\text{Equiv}(u, wa)$  the word  $u$  has been already added to  $V$ , and so the Learner has established that  $u \in \mathcal{L}(N)$ . So in particular the Teacher has answered positively a query about  $u$  and, due to the structure of workflow membership queries, it *has returned a run*  $uu_c$ , where  $u_c\mathbf{r}$  is a transition sequence leading back to the initial marking.

We prove that  $\text{Equiv}(x, y)$  holds if and only if the sequence  $xy_c$  is a run of  $N$ :



**Algorithm 1.** Learning the reachability graph**Output:** graph  $(V, E)$  isomorphic to the reachability graph of  $N$ 


---

```

 $V \leftarrow \emptyset; E \leftarrow \emptyset$ 
 $F \leftarrow \{\epsilon\}$  // queue of firing sequences
while not  $F.empty()$  do
   $w \leftarrow F.dequeue()$ 
  forall  $a \in T$  do
    if  $wa$  is accepted by the Teacher then
      /* This means  $wa \in \mathcal{L}(N)$  */
       $\sigma \leftarrow wa$ 
      forall  $u \in V$  do
        | if  $Equiv(u, wa)$  then  $\sigma \leftarrow u$ 
      end
      if  $\sigma = wa$  then  $F.enqueue(wa)$ 
      add  $\sigma$  to  $V$  and  $w \xrightarrow{a} \sigma$  to  $E$ 
    end
  end
end

```

---

**Proposition 1.** In Algorithm 1,  $Equiv(u, wa) = \mathbf{true}$  if and only if  $uw_c$  is a run of  $N$ , where  $waw_c$  is the run reported by the Teacher when positively answering the query about  $wa \in \mathcal{L}(N)$ .

*Proof.* If  $Equiv(u, wa) = \mathbf{true}$ , then there is a marking  $m$  such that  $m_0 \xrightarrow{u} m$  and  $m_0 \xrightarrow{wa} m$ . Because  $m_0 \xrightarrow{wa} m \xrightarrow{w_c r} m_0$ , we have  $m_0 \xrightarrow{u} m \xrightarrow{w_c r} m_0$ , which implies that  $uw_c$  is a run.

If  $u \cdot w_c$  is a run, then we have  $m_0 \xrightarrow{waw_c r} m_0$  and  $m_0 \xrightarrow{uw_c r} m_0$ . Let  $m$  be the marking such that  $m_0 \xrightarrow{wa} m$ . We then have  $m \xrightarrow{w_c r} m_0$ . Moreover,  $m$  is the only marking such that  $m \xrightarrow{w_c r} m_0$  (Petri nets are backward deterministic: given a firing sequence and its target marking, the source marking is uniquely determined). Since  $m_0 \xrightarrow{uw_c r} m_0$ , we then necessarily have  $m_0 \xrightarrow{u} m \xrightarrow{w_c r} m_0$ , and so in particular  $m_0 \xrightarrow{u} m$ . So both  $wa$  and  $u$  lead to the same marking  $m$ , and we have  $Equiv(u, wa) = \mathbf{true}$ .

We can now easily show that checking  $Equiv(u, wa)$  reduces to one single membership query.

**Proposition 2.** The check  $Equiv(u, wa)$  can be performed by querying whether  $uw_c \in \mathcal{L}(N)$ :  $Equiv(u, wa)$  holds if and only if the Teacher answers positively and returns the sequence  $uw_c$  itself as a run.

*Proof.* There are three possible cases:

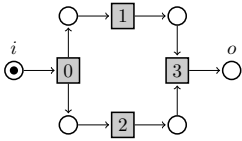
- The answer is negative.

Then  $uw_c \notin \mathcal{L}(N)$ , and so in particular it is not a run of  $N$ . So  $Equiv(u, wa) = \mathbf{false}$ .

- The answer is positive *and* the Teacher returns  $uw_c$  as run. Then  $\text{Equiv}(u, wa) = \mathbf{true}$  by Proposition [11](#)
- The answer is positive, but the Teacher returns  $uw_cv$  for some  $v \neq \epsilon$  as run. Since the Teacher returns a run  $uw_cv$  such that no proper prefix  $uw_cv'$  is a run, we have in particular by taking  $v' = \epsilon$  that  $uw_c$  is not a run. By Proposition [11](#) we have  $\text{Equiv}(u, wa) = \mathbf{false}$ .

*Remark 1.* In anticipation to the experiments described in Section [4](#), let us mention that in many cases the queries for  $uw_c$  do not even have to be submitted to the teacher (recall that  $wa$  labels the potentially new state and  $u$  labels a known state). Often we can deduce that  $uw_c$  is not fireable by observing that  $uw_c \notin L(A)$  where  $A$  is the part of the DFA that is already known. If we would query  $wau_c$  instead (which would also tell us if  $\text{Equiv}(u, wa) = \mathbf{true}$ ) we would not be able to discard any query because the neighbourhood of  $wa$  has not yet been explored. This is one of the reasons why this algorithm is so efficient in practice (cf. Section [4](#)).

*Example 1.* We now provide an example run of our algorithm, applied to the first part of the net in Figure [11](#). To simplify presentation we grouped together some queries which correspond to the interesting stages of the algorithm, i.e. after the teacher has answered a query with "yes". Furthermore we write  $w \cdot A$  for all queries  $wa$  with  $a \in A$ .



#	Query	Answer	Possible Automata
1-4	$\epsilon \cdot \{0, 1, 2, 3\}$	0(123)	$\overset{0}{\bullet} \xrightarrow{0} \bullet$
5	$\text{Equiv}(\epsilon, 0)?$ $\rightsquigarrow \epsilon \cdot 123$	no	$\bullet \xrightarrow{0} \bullet$
6-8	$0 \cdot \{0, 1, 3\}$	01(23)	$\overset{0}{\bullet} \xrightarrow{1} \bullet \xrightarrow{0} \overset{1}{\bullet}$ $\bullet \xrightarrow{0} \bullet \xrightarrow{1} \bullet$
9	$\text{Equiv}(\epsilon, 01)?$ $\rightsquigarrow \epsilon \cdot 23$	no	$\bullet \xrightarrow{0} \overset{1}{\bullet} \xrightarrow{0} \bullet \xrightarrow{1} \bullet$

10	$\text{Equiv}(0, 01)?$ $\rightsquigarrow 0 \cdot 23$	no	$\bullet \xrightarrow{0} \bullet \xrightarrow{1} \bullet$
11	02	02(13)	(4 Possibilities)
12-14	$\text{Equiv?}$ $(\{\epsilon, 0, 01\}, 02)$	no	$\bullet \xrightarrow{0} \bullet \begin{cases} \xrightarrow{1} \bullet \\ \xrightarrow{2} \bullet \end{cases}$
15-18	$01 \cdot \{0, 1, 2, 3\}$	012(3)	(5 Poss.)
19-22	$\text{Equiv?}$ $(\{\epsilon, 0, 01, 02\}, 012)$	no	$\bullet \xrightarrow{0} \bullet \begin{cases} \xrightarrow{1} \bullet \xrightarrow{2} \bullet \\ \xrightarrow{2} \bullet \end{cases}$
23-26	$02 \cdot \{0, 1, 2, 3\}$	021(3)	(6 Poss. - naive)
27	$[\text{Equiv}(021, 012)?]$	yes	$\bullet \xrightarrow{0} \bullet \begin{cases} \xrightarrow{1} \bullet \xrightarrow{2} \bullet \\ \xrightarrow{2} \bullet \xrightarrow{1} \bullet \end{cases}$
28-31	$012 \cdot \{0, 1, 2, 3\}$	0123( $\epsilon$ )	(7 Poss. - naive)
32	[Equiv-Queries]	no	$\bullet \xrightarrow{0} \bullet \begin{cases} \xrightarrow{1} \bullet \xrightarrow{2} \bullet \xrightarrow{3} \bullet \\ \xrightarrow{2} \bullet \xrightarrow{1} \bullet \end{cases}$

The "Answer"-column contains the run  $ww_c$  returned by the teacher, if  $w \in \mathcal{L}(N)$ —we put the continuations  $w_c$  in brackets. As observed in Remark [11](#), many queries (like " $\epsilon \cdot 23$ " in #9) do not really have to be asked—either because we already asked a prefix of the query that was rejected, or because the query is a prefix of a run supplied by the teacher and therefore we already know that it is accepted. We also do not need to ask query #27 because 021 and 012 have the same Parikh vector and therefore must lead to the same marking.

There is a technical issue that should be mentioned at this point. The algorithm delivers a net  $N'$  such that the reachability graphs of  $N$  and  $N'$  are isomorphic.

It follows that  $N'$  is reversible and bounded. However, we cannot guarantee that  $N'$  has the same bound as  $N$ . We consider this a minor problem, since  $N'$  and  $N$  are for behavioural purposes equivalent models.

**Complexity.** It follows clearly from the description of Algorithm [1](#) that the number of firing sequences added to the queue is equal to the number of reachable markings  $r$  of  $N$ . For the  $i$ -th sequence taken from the queue, say  $w$ , and for each transition, say  $a$ , we perform at most  $i$  membership queries: one to check if  $wa \in \mathcal{L}(N)$ , and at most  $(i - 1)$  for checks  $\text{Equiv}(u, wa)$ , because at that point  $V$  contains at most  $i - 1$  elements. So the algorithm performs at most  $\sum_{i=1}^r n \cdot i = nr(r + 1)/2 \in \mathcal{O}(n \cdot r^2)$  queries.

The following theorem sums up the results of the section.

**Theorem 3 (Learning by Exploration).** *We can learn a  $k$ -bounded and reversible net  $N$  with a number of workflow membership queries and a running time that are single exponential in the number of transitions of  $N$ .*

The proof follows easily from our discussion. The overall algorithm, that we call WNL, uses the learning technique of Section [3.3](#) to learn a minimal DFA  $A$  such that  $\mathcal{L}(A) = \mathcal{L}(N)$ . Section [3.1](#) shows that  $A$  is single exponential in the number of transitions of  $N$ , and so it can be learned with a single exponential number of queries. Section [3.2](#) shows that this minimal DFA is (isomorphic to) the reachability graph of  $N$ . We can then apply the polynomial algorithm of [\[BBD95\]](#) for synthesis up to isomorphism (S2).

A final question is what happens if the Teacher's answers are not compatible with any  $k$ -bounded and reversible net  $N$ . In this case there are two possibilities: they are not compatible with any minimal DFA having at most  $(k + 1)^n$  states, or they are compatible with some such DFA, but this DFA is not the marking-DFA of any net. In the first case the algorithm can stop when the number of generated states exceeds  $(k + 1)^n$ . In the second case, the algorithm terminates and produces a DFA, but the synthesis algorithm of [\[BBD95\]](#) does not return a net.

### 3.4 Mixing Process Mining and Learning

The algorithm we have just presented does not assume the existence of an event log: the Learner only gets information from membership queries. However, as explained in the introduction, we consider our learning approach as a way of complementing log-based process mining. In this section we explain how to modify the algorithm accordingly.

We assume the existence of an event log consisting of use cases. Given the set of tasks  $T$  of the business process, we can think of each use case as a word  $w \in T^*$ , such that  $w$  corresponds to a run of the reversible net to be learnt. The event log then corresponds to a language  $L \subseteq T^*$ .

In a first step we construct a minimal DFA for the language  $L$ . This can be done space-efficiently in a number of ways. For instance, we can divide the set of

runs in two halves  $L_1, L_2$ , recursively compute minimal DFAs  $A_1, A_2$  recognizing  $L_1$  and  $L_2$ , and then compute the minimal DFA for  $L$  from  $A_1, A_2$  using an algorithm very similar to the one that computes the union of two binary decision diagrams [And99]. Once this is done, we easily get the minimal DFA  $A$  for the language of prefixes of  $(Lr)^*$  (this requires to add one extra state and make all states final).

Once  $A$  is computed, we assign to each state  $q$  of  $A$  a word  $w_q$  such that  $q_0 \xrightarrow{w_q} q$ . For every two states  $q_1, q_2$ , we check whether the states correspond to the same reachable marking by calling  $\text{Equiv}(w_{q_1}, w_{q_2})$ . After this step we are in the same situation we would have reached if the algorithm would have queried all the words  $w_q$ .

From a practical point of view, notice that it is very inefficient to ask the Teacher for each pair of states  $q_1, q_2$  whether  $\text{Equiv}(w_{q_1}, w_{q_2})$ . A better procedure is to ask the Teacher, given a sequence  $w$ , which are the letters  $a$  such that  $wa$  can be extended to a use case. We call them the *possible extensions* of  $w$ . The test  $\text{Equiv}(w_{q_1}, w_{q_2})$  need only be carried out for sequences  $w_{q_1}, w_{q_2}$  having the same set of extensions. Note that the teacher does not have to provide full runs for any of these possible extensions so this is quite a simple task.

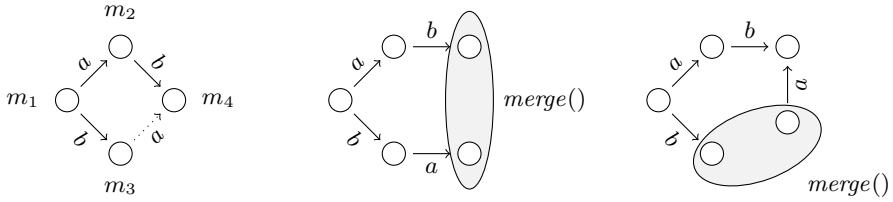
We can even further reduce the number of calls to  $\text{Equiv}()$  by first merging states for which we can already deduce that they have to be equivalent. Some criteria, which are easy properties of Petri nets, and can be directly used to trim a DFA that was generated from event logs are:

- The DFA is backward deterministic: if  $m_1 \xrightarrow{a} m_3$  and  $m_2 \xrightarrow{a} m_3$  for some  $a \in T$  then  $m_1 = m_2$
- If two words  $w_1, w_2$  only differ in the order of their letters (i.e. their Parikh vectors coincide  $P(w_1) = P(w_2)$ ) then they lead to the same state
- Given a  $k$ -bounded net  $N$ , if  $vw^{k+1} \in \mathcal{L}(N)$  for some words  $v, w$  then  $w$  describes a cycle in the reachability graph of  $N$

A further criterion for *pure nets* is the “diamond property”: We can add transitions that have to be present due to basic Petri net properties. A *diamond* is a subgraph in the reachability graph of a net with four states that are connected in the following way:  $m_1 \xrightarrow{a} m_2, m_1 \xrightarrow{b} m_3, m_2 \xrightarrow{b} m_4, m_3 \xrightarrow{a} m_4$ . A diamond is *incomplete* if it is missing exactly one transition (see Figure 2). One can easily see that incomplete diamonds can always be completed with the missing transition (in the case of pure nets), i.e., if an incomplete diamond is found in the DFA, we can add the missing transition. This *diamond property* can also be used to merge states as indicated in Figure 2.

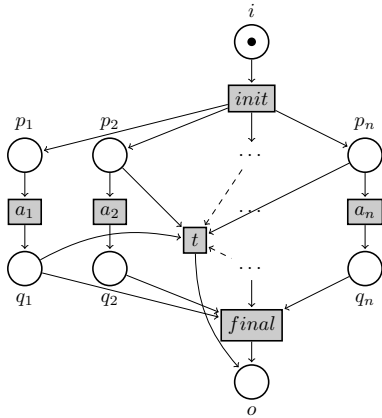
### 3.5 A Lower Bound for Petri Net Learning

We now show that we cannot in general solve the learning problem in subexponential time, by providing a hard-to-learn instance. We will show with the help of an adversary argument that any learning algorithm has to ask at least  $\Omega(2^n)$  membership queries to derive the correct net, where  $n$  is the number of transitions.



**Fig. 2.** Incomplete diamond (left), states merged because of equal parikh-vectors (middle) or by using the diamond property (middle and right)

Consider the following set  $\mathcal{N}$  of workflow nets. All the nets in  $\mathcal{N}$  have the same number  $n + 3$  of transitions: two transitions *init* and *final*, transitions called  $a_1, \dots, a_n$ , and a transition  $t$  (see Figure 3). The pre- and postsets of all transitions but  $t$ , which are identical for all nets of  $\mathcal{N}$ , are shown in the figure. The postset of  $t$  is always the place  $o$ . The preset of  $t$  always contains for each  $i$  exactly one of the places  $p_i$  or  $q_i$ , and the only difference between two nets in  $\mathcal{N}$  is their choice of  $p_i$  or  $q_i$ . Clearly, the set  $\mathcal{N}$  contains  $2^n$  workflow nets, all of them sound.



**Fig. 3.** Hard-to-learn instance for Petri net learning

always rules out *exactly one* of the nets of  $\mathcal{N}$ , namely the one in which  $\bullet t = S$ . The worst case appears when the Learner ask queries “in the worst possible order”, eliminating all nets of  $\mathcal{N}$  but the right one. This requires  $2^n - 1$  queries.

### 4 Practical Experiences

To get insights in the practical feasibility of the derived algorithm WNL, we have developed a prototype learning and synthesis tool for workflow nets and examined its practical performance on a number of examples.

For each net  $N \in \mathcal{N}$  there is exactly one subset of  $\{a_1, \dots, a_n\}$  such that  $t$  can fire after the transitions of the set have fired. We call this subset  $S_N$ . Notice that if we know  $S_N$  then we can infer  $\bullet t$ .

We ease the task for the Learner by assuming she knows that the net to be learned belongs to  $\mathcal{N}$ . Her task consists only of finding out  $\bullet t$ , or, equivalently, the set  $S_N$ .

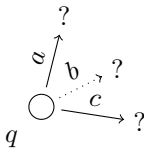
A query of an optimal Learner has always the form  $a_{i_1} a_{i_2} \dots a_{i_k} t$ , because querying any  $a_i$  after  $t$  does not provide the Learner with any information. Furthermore the order of the  $a_i$  is not important—all these transitions are independent and the Learner already knows this. So we can view a query just as a subset  $S$  of the set of all transitions. A negative answer to a query  $S$  always

**Implementation.** Our prototype is written in C++ with approximately 3,000 lines of code and uses LIBALF for dealing with automata. LIBALF is part of the automata learning factory currently developed jointly at RWTH Aachen and TU München<sup>1</sup> [BKK<sup>+</sup>10].

The synthesis algorithm (S2) of [BBD95] is implemented using the LP\_SOLVE<sup>2</sup> framework to efficiently solve the linear programs needed for computing the places of the net. Furthermore LP\_SOLVE is used for eliminating redundant places after the net has been synthesized to reduce its size and to make it look more appealing. The implementation is currently not tailored to user interaction but consults pre-existing workflow nets for queries. Outputs are given in form of dot-files that can be visualized using the GRAPHVIZ toolkit.

**Experimental Results.** We tested our implementation on various examples of pure, safe and reversible nets. The examples range from existing sound workflow nets obtained in case studies performed by [Ver04] to more standard examples like mutual exclusion between processes and an  $n$ -cell buffer with  $2^n$  reachable markings. The latter example is especially suitable to understand scalability issues of the algorithms. The "absence" workflow is loosely modelled after an example from [SAP01], the "complaint" workflow is the example presented in our background section (Figure 1).

We applied our implementation once without any event logs as initial knowledge and then again with randomly generated logs as input and counted the number of queries needed to learn the model. Besides counting the queries needed for `Equiv()`, we only count queries answered positively by the Teacher, as these correspond to runs supplied by him, and thus reflect the actual work to be done by an expert in an adequate manner.



**Fig. 4.** Querying extensions at state  $q$ , possible extensions: solid arrows

To illustrate this, consider the task of learning the sequence of calendar months: instead of asking twelve questions of the form “Does January, February, ... come after July?” (we call these “small-step” queries) we would just ask “Which month comes after July?”. So we count every continuation provided by the teacher as one query. In the situation of Figure 4 we would count 2 workflow membership queries compared to 3 “small-step” queries. We have also included the number of “small-step” queries in the table below for comparison.

We have first collected the number of membership queries needed by WNL when learning a model “from scratch” with respect to the size of the alphabet and the reachability graph, see Figure 5. On the chosen examples, the number of membership queries ranges between 12 and 3100. The series of the  $n$ -cell buffer examples from  $n = 2$  to  $n = 8$  suggests that the practical performance of WNL is even better than quadratic in the number of reachable markings.

<sup>1</sup> <http://libalf.informatik.rwth-aachen.de/>

<sup>2</sup> <http://lpsolve.sourceforge.net/>

Model	$ T $	$ RG $	ssq	WNL
buf_2	3	4	19	12
buf_3	4	8	52	32
buf_4	5	16	137	85
buf_5	6	32	344	216
buf_6	7	64	842	538
buf_7	8	128	2008	1304
buf_8	9	256	4707	3107
mutex_2	6	8	74	40
mutex_3	9	20	300	168
mutex_4	12	48	1026	594
order_simp	9	7	77	23
absence	11	8	109	32
complaint	12	11	155	37
transit1	25	77	2256	474

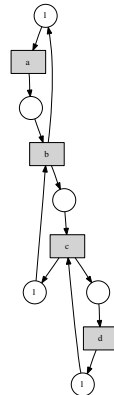
**Fig. 5.** Membership queries needed by WNL *without any event logs*; ssq = number of “small-step” queries, RG = reachability graph

of the larger models with logs of different sizes (see Figure 7).<sup>3</sup> We observe that already quite small logs drastically reduce the number of queries to be answered. At the same time, because our logs may not contain unique but many identical entries, larger logs contribute less and less new knowledge. This reflects the situation for real-life logs, which mostly contain common executions of a

Next, we studied the effect of learning workflow nets in the presence of existing logs. To this end, we used our tool to generate random event logs containing a varying number of runs (see Figure 6 for an example log). The runs in the generated log-files are not unique—runs that are more likely will probably appear multiple times, which is also the case for real-world event logs. For the random logs we calculated the average number of queries over 100 executions.

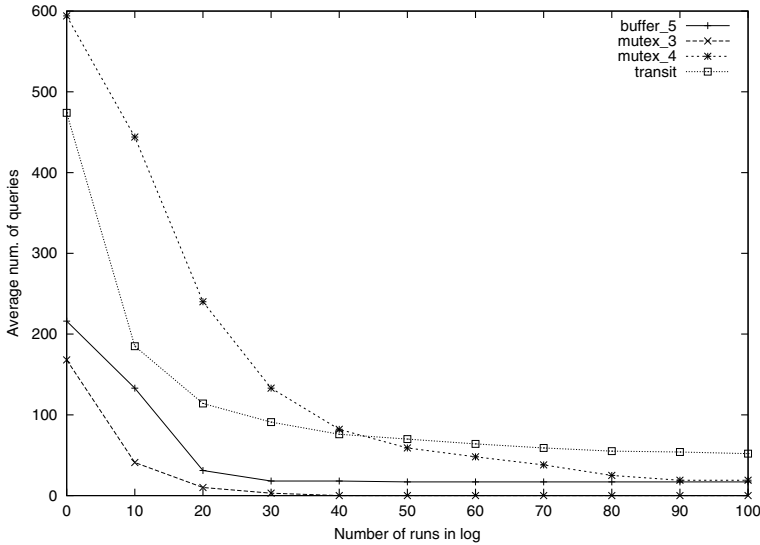
We found out that for tiny models like the buffer with two cells or the “complaint” workflow a very small number of runs ( $< 10$ ) suffices to already construct the model. The Teacher does not have to supply additional runs for these. Clearly, for larger models, we can only expect that the Teacher’s work is reduced but not completely eliminated when logs are given. To illustrate the impact of event logs on the learning process we show how the number of queries behaves for some

- . a . b . a . c . d . b . c . a . d . b . c . d .
- . a . b . c . d .
- . a . b . c . a . b . d . c . a . d . b . c . d .
- . a . b . a . c . d . b . c . d .
- . a . b . a . c . b . a . d . c . d . b . c . d .
- . a . b . c . a . d . b . a . c . d . b . c . d .
- . a . b . c . a . b . d . c . d .
- . a . b . a . c . b . d . a . c . d . b . c . d .
- . a . b . c . d .
- . a . b . c . d .



**Fig. 6.** Example event log for 3-cell buffer

<sup>3</sup> Also larger examples behave in the same way, yet, we depicted models requiring a number of queries in the same order of magnitude to optimize the figure.



**Fig. 7.** Average number of queries needed by WNL applied to event logs of different sizes

workflow but lack less common runs. In other words, it seems most promising for practical applications, to combine knowledge from (small) logs with that of Teachers responsible for “corner cases” to actually learn the workflow net in question efficiently.

The time needed for learning the nets in an applied setting is of course dominated by the number of queries a user has to answer. Synthesizing the resulting Petri net using the method proposed by Darondeau et al. (see Section 2) together with some post-processing to remove redundant places needs just a few seconds in the worst case and is therefore negligible.

The results depicted in Figures 5 and 7 suggest that, despite the seemingly intimidating result in Section 3.5, learning of workflow models is quite feasible for practical applications.

## 5 Conclusion

We have presented a new approach for mining workflow nets based on learning techniques. The approach palliates the problem of incompleteness of event logs: if a log is incomplete, our algorithm derives membership queries identifying the missing knowledge. The queries can be passed to an expert, whose answers allow to produce a model.

We have shown the correctness and completeness of our approach given a teacher answering workflow membership queries. Starting with general combinatorial arguments showing that workflow models can in principle be learned,



we have derived a learning algorithm requiring a single exponential number of queries in the worst case, and we have given a matching lower bound. We have also shown experimental evidence indicating that the combination of an event log, even of small size, and a Teacher responsible for providing information on “corner cases” allows to efficiently produce models in practically relevant cases.

There are several promising paths for further research. One aspect is the application of learning to the *design* of workflows. In this approach an expert on business processes and a modelling expert (or an adequate software) cooperate. The modelling expert asks queries about how the workflow should behave, which are answered by the Teacher, until a model accepted by the business process expert is produced. We expect to transfer ideas from the field of learning process models of software systems [BKKL09] to workflow systems, and develop “teaching assistants” that filter the queries, automatically answering those for which the answer can be deduced from current information (for instance because it is known that two tasks must be concurrent), and only passing to the expert the remaining ones. Here we expect to profit from related work by Desel, Lorenz and others [BDML09]. An important point for process mining and even more for process design is designing fault tolerance techniques allowing to cope with false answers by the Teacher. Finally, learning more general classes of Petri nets, and applications to modelling/reconstruction of distributed systems, or biological/chemical processes, are also promising paths for future work.

## References

- [AGL98] 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)
- [And99] Andersen, H.R.: An introduction to binary decision diagrams. Technical report (1999), <http://www.itu.dk/people/hra/bdd-eap.pdf>
- [Ang87] Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* 75(2), 87–106 (1987)
- [BBD95] Badouel, E., Bernardinello, L., Darondeau, P.: Polynomial algorithms for the synthesis of bounded nets. In: Mosses, P.D., Schwartzbach, M.I., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915. Springer, Heidelberg (1995)
- [BDBM96] Badouel, E., Darondeau, P.: On the synthesis of general petri nets. Technical report, INRIA (1996)
- [BDKM08] Bergenthum, R., Desel, J., Kölbl, C., Mauser, S.: Experimental results on process mining based on regions of languages. In: CHINA 2008, Workshop at the Applications and Theory of Petri Nets: 29th International Conference (2008)
- [BDLM07] Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007)

- [BDML09] Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Construction of process models from example runs. *T. Petri Nets and Other Models of Concurrency* 2, 243–259 (2009)
- [BKKL09] Bollig, B., Katoen, J.-P., Kern, C., Leucker, M.: Learning communicating automata from MSCs. *IEEE Transactions on Software Engineering, TSE* (in press, 2009)
- [BKK<sup>+</sup>10] Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.: libalf: the Automata Learning Framework. In: *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV 2010)*. LNCS. Springer, Heidelberg (to appear, 2010)
- [Cho78] Chow, T.S.: Testing software design modeled by finite-state machines. *TSE* 4(3), 178–187 (1978); Special collection based on COMPSAC
- [KRS06] Kindler, E., Rubin, V., Schäfer, W.: Process mining and petri net synthesis. In: Eder, J., Dustdar, S. (eds.) *BPM Workshops 2006*. LNCS, vol. 4103, pp. 105–116. Springer, Heidelberg (2006)
- [RGvdA<sup>+</sup>07] Rubin, V., Günther, C.W., van der Aalst, W.M.P., Kindler, E., van Dongen, B.F., Schäfer, W.: Process mining framework for software processes. In: Wang, Q., Pfahl, D., Raffo, D.M. (eds.) *ICSP 2007*. LNCS, vol. 4470, pp. 169–181. Springer, Heidelberg (2007)
- [SAP01] SAP AG. *SAP Business Workflow Demo Examples (BC-BMT-WFM)* (2001)
- [Vas73] Vasilevski, M.P.: Failure diagnosis of automata. *Cybernetic* 9(4), 653–665 (1973)
- [vdA98] van der Aalst, W.M.P.: The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers* 8(1), 21–66 (1998)
- [vdAvDG<sup>+</sup>07] van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., de Medeiros, A.K.A., Rozinat, A., Rubin, V., Song, M. (Eric)Verbeek, H.M.W., Weijters, A.J.M.M.: Prom 4.0: Comprehensive support for *real* process analysis. In: Kleijn, J., Yakovlev, A. (eds.) *ICATPN 2007*. LNCS, vol. 4546, pp. 484–494. Springer, Heidelberg (2007)
- [vdAvDH<sup>+</sup>03] 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 Knowl. Eng.* 47(2), 237–267 (2003)
- [vdAvH04] van der Aalst, W., van Hee, K.: *Workflow Management. Models, Methods, and Systems*. MIT Press, Cambridge (2004)
- [Ver04] Verbeek, H.M.W.: *Verification of WF-nets*. PhD thesis, Technische Universiteit Eindhoven (2004)

# Process Mining from a Basis of State Regions

Marc Solé<sup>1</sup> and Josep Carmona<sup>2</sup>

<sup>1</sup> Computer Architecture Department, UPC  
msole@ac.upc.edu

<sup>2</sup> Software Department, UPC  
jcarmona@lsi.upc.edu

**Abstract.** A central problem in the area of Process Mining is to obtain a formal model that represents selected behavior of a system. The theory of regions has been applied to address this problem, enabling the derivation of a Petri net whose language includes a set of traces. However, when dealing with real-life systems, the available tool support for performing such task is unsatisfactory, due to the complex algorithms that are required. In this paper, the theory of regions is revisited to devise a novel technique that explores the space of regions by combining the elements of a region basis. Due to its light space requirements, the approach can represent an important step for bridging the gap between the theory of regions and its industrial application. Experimental results improve in orders of magnitude state-of-the-art tools for the same task.

## 1 Introduction

Nowadays the formal reasoning of a system is sometimes restricted by the difficulty of having a formal model that describes its behavior. This problem may appear at several stages of the life cycle: design, verification, and optimization. Aware of the problem, some companies have started to incorporate tools to discover formal models from executions of a system. This was the driving force that originated the area of *Process Mining*, where the goal is to obtain a formal model (e.g., a Petri net) that includes the behavior of a system. In this work we present a novel strategy for this problem.

The *synthesis problem* [1] is related to mining: it consists in building a Petri net that has a behavior equivalent to a given transition system. The problem was first addressed by Ehrenfeucht and Rozenberg [2] introducing *regions* to model the sets of states that characterize marked places. In the area of synthesis, some techniques have been proposed to take the theory of regions in to practice. In [3] polynomial algorithms for the synthesis of bounded nets were presented. These algorithms have been adapted for the problem of process mining in [4]. In [5], the theory of regions was applied for the synthesis of safe Petri nets with bisimilar behavior. Recently, the theory in [5] has been extended to bounded Petri nets [6].

Mining differs from synthesis in the knowledge assumption: while in synthesis one assumes a complete description of the system, only a partial description of the system is assumed in mining. However, synthesis can be adapted for

mining in two ways: either the initial set of traces (called *log*) is encoded as a transition system (introducing state information, as described in [7]) and state-based methods for mining [8] are applied, or language-based methods are used directly on the log [4,9]. In this paper we follow the first approach.

Due to its complexity, the theory of regions might become impractical for large inputs. In this paper, we present methods to alleviate significantly the complexity of the region-based approach. The main idea is based on the observation that the set of regions necessary for deriving a Petri net might be obtained by linear combinations of a small set of regions, i.e., from a *basis* of regions. This technique deviates from previous state-based methods for computing regions [6,8], where the full lattice of multisets of states was explored to find the regions. The main contributions of this paper are:

- *Methods to efficiently compute a basis of regions*, based on the isomorphism between the structural and state-based representation of regions. Moreover, when the input transition system is derived from a language, the obtention of a basis is shown to be simplified.
- *An algorithm to explore the region space*, that efficiently searches for minimal regions using a very simple criterion to determine if a region is guaranteed to be non-minimal.
- The theory of this paper has been implemented in a tool [10]. The experiments demonstrate the capacity of handling systems for which related approaches fail. Moreover, for well-known benchmarks, the quality of the derived results is shown to be similar to the one obtained for related approaches.

**Organization.** We start by giving the necessary background in Sect. 2. Methods to compute a region basis are presented in Sect. 3, and Sect. 4 provides a strategy to explore the space of regions from a region basis to derive a Petri net. Experiments and related work are presented in Sect. 5 and Sect. 6, respectively<sup>1</sup>.

## 2 Background

### 2.1 Finite Transition Systems and Petri Nets

**Definition 1 (Transition system).** A transition system (*TS*) is a tuple  $\langle S, \Sigma, T, s_0 \rangle$ , where  $S$  is a set of states,  $\Sigma$  is an alphabet of actions,  $T \subseteq S \times \Sigma \times S$  is a set of (labelled) transitions, and  $s_0 \in S$  is the initial state.

We use  $s \xrightarrow{e} s'$  as a shortcut for  $(s, e, s') \in T$ , and we denote its transitive closure as  $\xrightarrow{*}$ . A state  $s'$  is said to be *reachable from state*  $s$  if  $s \xrightarrow{*} s'$ . We extend the notation to transition sequences, i.e.,  $s_1 \xrightarrow{\sigma} s_{n+1}$  if  $\sigma = e_1 \dots e_n$  and  $(s_i, e_i, s_{i+1}) \in T$ . We denote  $\#(\sigma, e)$  the number of times that event  $e$  occurs in  $\sigma$ . Let  $A = \langle S, \Sigma, T, s_0 \rangle$  be a TS. We consider connected TSs that satisfy the following axioms: i)  $S$  and  $\Sigma$  are finite sets, ii) every event has an occurrence and iii) every state is reachable from the initial state. The *language* of a TS  $A$ ,  $\mathcal{L}(A)$ , is the set of traces feasible from the initial state.

<sup>1</sup> An extended version of this paper including all the formal proofs can be found in [11].

**Definition 2 (Petri net [12]).** A Petri net (PN) is a tuple  $(P, T, W, M_0)$  where the sets  $P$  and  $T$  represent finite sets of places and transitions, respectively, and  $W : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$  is the weighted flow relation. The initial marking  $M_0 \in \mathbb{N}^{|P|}$  defines the initial state of the system.

A transition  $t \in T$  is *enabled* in a marking  $M$  if  $\forall p \in P : M[p] \geq W(p, t)$ . Firing an enabled transition  $t$  in a marking  $M$  leads to the marking  $M'$  defined by  $M'[p] = M[p] - W(p, t) + W(t, p)$ , for  $p \in P$ , and is denoted by  $M \xrightarrow{t} M'$ . The set of all markings reachable from the initial marking  $M_0$  is called its Reachability Set. The *Reachability Graph* of  $N$ , denoted  $\text{RG}(N)$ , 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  $M_1 \xrightarrow{t} M_2$ . We use  $\mathcal{L}(N)$  as a shortcut for  $\mathcal{L}(\text{RG}(N))$ .

### 2.2 Generalized Regions

The theory of regions [21] provides a way to derive a Petri net from a transition system. Intuitively, a region corresponds to a place in the derived Petri net. In the initial definition, a region was defined as a subset of states of the transition system satisfying an homogeneous relation with respect to the set of events. Later extensions [13, 14, 6] generalize this definition to multisets, which is the notion used in this paper.

**Definition 3 (Multiset,  $k$ -bounded Multiset, Subset).** Given a set  $S$ , a multiset  $r$  of  $S$  is a mapping  $r : S \rightarrow \mathbb{Z}$ . The number  $r(s)$  is called the multiplicity of  $s$  in  $r$ . Multiset  $r$  is  $k$ -bounded if all its multiplicities are less or equal than  $k$ . Multiset  $r_1$  is a subset of  $r_2$  ( $r_1 \subseteq r_2$ ) if  $\forall s \in S : r_1(s) \leq r_2(s)$ .

We define the following operations on multisets:

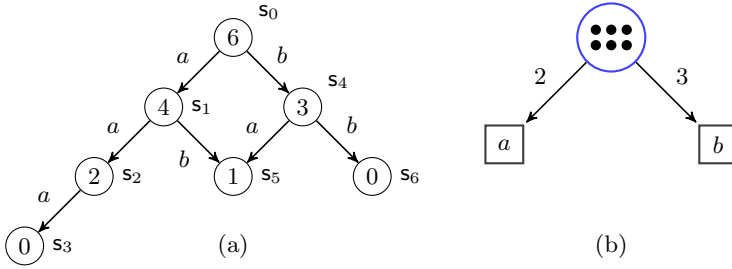
**Definition 4 (Multiset operations)**

$$\begin{array}{ll}
 \text{Maximum power } \text{pow}(r) &= \max_{s \in S} r(s) \\
 \text{Minimum power } \text{minp}(r) &= \min_{s \in S} r(s) \\
 \text{Scalar product } (k \cdot r)(s) &= k \cdot r(s), \text{ for } k \in \mathbb{Z} \\
 \text{Scalar sum } (r + k)(s) &= r(s) + k, \text{ for } k \in \mathbb{Z} \\
 \text{Sum } (r_1 + r_2)(s) &= r_1(s) + r_2(s) \\
 \text{Subtraction } (r_1 - r_2)(s) &= r_1(s) - r_2(s)
 \end{array}$$

The operations described above have algebraic properties, e.g.,  $r + r = 2 \cdot r$  and  $r_1 - k \cdot r_2 = r_1 + (-k) \cdot r_2$ .

**Definition 5 (Gradient).** Let  $\langle S, \Sigma, T, s_0 \rangle$  be a TS. Given a multiset  $r$  and a transition  $s \xrightarrow{e} s' \in T$ , its gradient is defined as  $\delta_r(s \xrightarrow{e} s') = r(s') - r(s)$ . If all the transitions of an event  $e \in \Sigma$  have the same gradient, we say that the event  $e$  has constant gradient, whose value is denoted as  $\delta_r(e)$ .

**Definition 6 (Region).** A region  $r$  is a multiset defined in a TS, in which all the events have constant gradient.



**Fig. 1.** (a) Region in a TS:  $r(s_0) = 6, r(s_1) = 4, \dots, r(s_6) = 0$ , (b) corresponding place in the Petri net

*Example 1.* Fig. 1(a) shows a TS. The numbers within the states correspond to the multiplicity of the multiset  $r$  shown. Multiset  $r$  is a region because both events  $a$  and  $b$  have constant gradient, i.e.  $\delta_r(a) = -2$  and  $\delta_r(b) = -3$ . There is a direct correspondence between regions and places of a PN. The gradient of the region describes the flow relation of the corresponding place, and the multiplicity of the initial state indicates the number of initial tokens [6]. Fig. 1(b) shows the place corresponding to the region shown in Fig. 1(a).

We say that region  $r$  is *normalized* if  $\minp(r) = 0$ . Any region  $r$  can become normalized by subtracting  $\minp(r)$  to the multiplicity of all the states:

**Definition 7 (Normalization).** We denote by  $\downarrow r$  the normalization of a region  $r$ , so that  $\downarrow r = r - \minp(r)$ .

It is useful to define a normalized version of the sum operation between regions, since it is closed in the class of normalized regions.

**Definition 8 (Normalized sum).** Let  $r_1$  and  $r_2$  be normalized regions, we denote by  $r_1 \oplus r_2$  their normalized sum, so that  $r_1 \oplus r_2 = \downarrow(r_1 + r_2)$ .

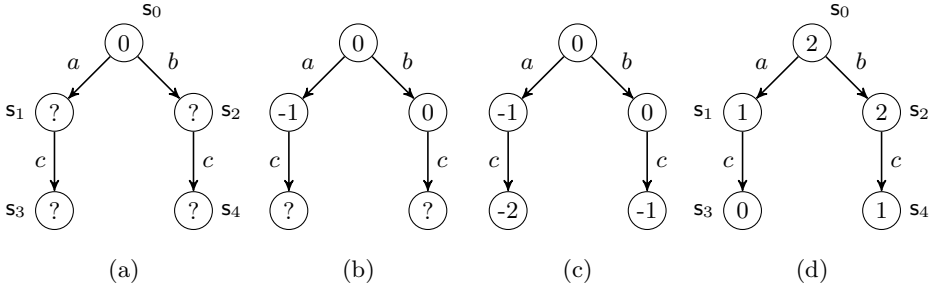
**Definition 9 (Gradient vector).** Let  $r$  be a region of a TS whose set of events is  $\Sigma = \{e_1, e_2, \dots, e_n\}$ . The gradient vector of  $r$ , denoted as  $\Delta(r)$ , is the vector of the event gradients, i.e.  $\Delta(r) = (\delta_r(e_1), \delta_r(e_2), \dots, \delta_r(e_n))$ .

**Proposition 1.** Gradient vectors have the following properties:

$$\begin{aligned} \Delta(r_1 + r_2) &= \Delta(r_1) + \Delta(r_2) & \Delta(k \cdot r) &= k \cdot \Delta(r) \\ \Delta(r + k) &= \Delta(r) & \Delta(r_1 - r_2) &= \Delta(r_1) - \Delta(r_2) \\ \Delta(r_1 \oplus r_2) &= \Delta(r_1) + \Delta(r_2) \end{aligned}$$

Regions can be partitioned into classes using  $\Delta(r)$ :

**Definition 10 (Canonical region).** Two regions  $r_1$  and  $r_2$  are said to be equivalent if their gradient is the same, i.e.  $r_1 \equiv r_2 \Leftrightarrow \Delta(r_1) = \Delta(r_2)$ . Given a region  $r$ , the equivalence class of  $r$ , is defined as  $[r] = \{r_i \mid r_i \equiv r\}$ . A canonical region is the normalized region of an equivalence class, i.e.  $\downarrow r$ .



**Fig. 2.** Obtaining the region with gradient  $(-1, 0, -1)$  in a TS, using a breadth-first search. (a) A zero multiplicity is assigned to the initial state. (b,c) Multiplicities after the first and second iterations, respectively. Some of the multiplicities are negative. To normalize them, the minimum power, in this case -2, is subtracted to all states, yielding the region in (d).

An example of canonical region is provided in Fig. 3(b), where a TS is shown in which some regions have been shadowed. The canonical region  $r_1 = \{s_1, s_2\}$  has gradient vector  $\Delta(r_1) = (+1, +1, -1)$ . Under some conditions, the set of minimal canonical regions is enough to guarantee some equivalence between the TS and the derived PN [1].

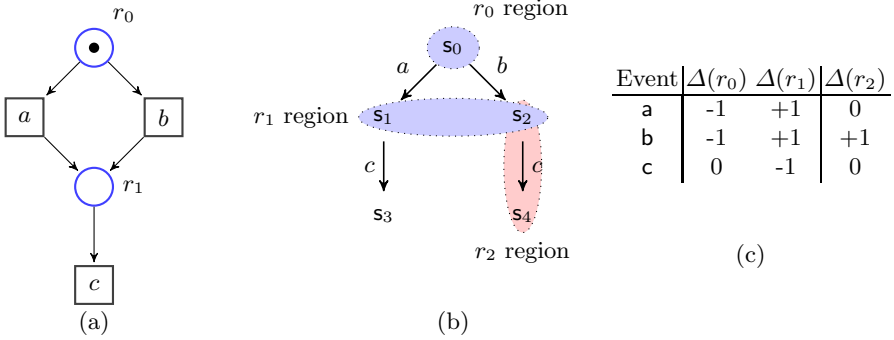
**Definition 11 (Subregion, Empty region, Minimal canonical region).**  $r_1$  is a subregion of  $r_2$ , denoted as  $r_1 \sqsubseteq r_2$ , if, for any state  $s$ ,  $\downarrow r_1(s) \leq \downarrow r_2(s)$ . We denote by  $\emptyset$  the region in which all states have zero multiplicity. A minimal canonical region  $r$  satisfies that for any other region  $r'$ , if  $r' \sqsubseteq r$  then  $r' \equiv \emptyset$ .

### 2.3 Derivation of Regions from Gradient Vectors

A region corresponding to a gradient vector (c.f. Def. 9) can be obtained by traversing the TS from the initial state, with an arbitrary multiplicity (0 for instance), and giving a multiplicity to each discovered state based on the multiplicity of the source state of any incoming arc and the gradient of the event that labels the arc. To obtain a normalized region, the smallest multiplicity computed during the exploration (i.e. the minimum power of the region) is stored, and then subtracted to all multiplicities. An example is shown in Fig. 2.

## 3 Finding a Region Basis

The goal of this section is to obtain a *basis* of regions, i.e. a set of linearly independent regions  $B$  that can represent any other region by linear combinations of the elements in  $B$ . In general, the size of  $B$  is significantly smaller than the number of minimal canonical regions (see Theorem 1 below). The results of this section are grounded in the theory presented in [15].



**Fig. 3.** PN whose places,  $r_0$  and  $r_1$ , have regions that cannot produce, by linear combination, some of the regions present in its RG, for instance region  $r_3$

**Definition 12 (Region basis).** *Given a TS, a region basis  $B = \{r_1, r_2, \dots, r_n\}$  is a minimal subset of the canonical regions of TS such that any region  $r$  can be expressed as a linear combination of the elements in  $B$  (i.e.  $r = \sum_{i=1}^n c_i \cdot r_i$ , with  $c_i \in \mathbb{Z}$ ,  $r_i \in B$ ).*

The set of canonical regions of a TS, together with the normalized sum operation ( $\oplus$ ), forms a free Abelian group [14]. Consequently, there exists a *basis* (i.e. subset of the group) such that every element in the group can be rewritten as a unique linear combination of the basis elements. In particular all the minimal canonical regions can be generated from the basis. As the following theorem states, the size of such a basis is bounded:

**Theorem 1 ([14]).** *The size of a region basis for TS  $A = \langle S, \Sigma, T, s_0 \rangle$  is less or equal to  $\min(|\Sigma|, |S| - 1)$ .*

*Example 2.* In TS of Fig. 3(b), the set of minimal canonical regions is formed by  $r_0 = \{s_0\}$ ,  $r_1 = \{s_1, s_2\}$ ,  $r_2 = \{s_2, s_4\}$ ,  $r_3 = \{s_1, s_3\}$ ,  $r_4 = \{s_3, s_4\}$ . However, we can express  $r_3$  and  $r_4$  in terms of the other regions:  $r_3 = -r_0 - r_2$  and  $r_4 = -r_0 - r_1$ . Note that, without normalizing the resulting regions it might be difficult to see the equivalence. For instance  $-r_0 - r_1 = \{-s_0, -s_1, -s_2\}$  which requires to subtract  $-1$  (add 1) to each state multiplicity to obtain a normalized region, thus  $\{-s_0, -s_1, -s_2\} + 1 = \{s_3, s_4\} = r_4$ . Since any region can be expressed as a sum of minimal canonical regions [14], and  $r_3$  and  $r_4$  are linear combinations of the other regions, a possible basis is formed by only three regions (as there are only three events), namely  $r_0, r_1$  and  $r_2$ , whose gradient vectors appear in Fig. 3(c).

In the previous example we have found a basis from the set of minimal canonical regions. In Sect. 4 the opposite process will be performed: from a basis, obtain a set of minimal canonical regions. What remains in this section is to present methods to obtain a basis.



An efficient method to compute a region basis without requiring the set of minimal canonical regions can be devised if we use the following observation: a set of regions whose corresponding gradient vectors form a basis of the universe of gradient vectors also forms a basis of the universe of regions.

**Proposition 2.** *Given a TS  $A$ , let  $CGR_A$  denote the set of canonical regions of  $A$  and  $D_A$  be the set of their gradient vectors. If  $B_\Delta = \{d_1, d_2, \dots, d_n\} \subseteq D_A$  is a basis of the group  $(D_A, +)$ , then  $B = \{r_1, r_2, \dots, r_n\} \subseteq CGR_A$  such that  $\Delta(r_i) = d_i$  is a basis for the group  $(CGR_A, \oplus)$ .*

*Proof.* To prove that  $B$  is a basis, two properties must be shown. First, any region has to be expressible as a linear combination of the elements in  $B$ . The  $\Delta$  function establishes an isomorphism between the free Abelian groups  $(CGR_A, \oplus)$  and  $(D_A, +)$ . Thus any gradient vector in  $D_A$  can be expressed as  $\sum_i c_i d_i$ , which is the gradient vector of the region  $\bigoplus_i c_i r_i$ . Since any normalized region in  $A$  has its gradient vector in  $D_A$ , and any such vector is the gradient vector of a region  $\bigoplus_i c_i r_i$ , then any region in  $CGR_A$  can be generated as a linear combination of the elements in  $B$ . The second required property for  $B$  to be a basis is that the elements in  $B$  are linearly independent. Since both groups  $(CGR_A, \oplus)$  and  $(D_A, +)$  are isomorphic, their basis has the same rank (*i.e.* have the same number of elements), implying that all elements in  $B$  are linearly independent.  $\square$

Hence, a region basis can be found by: (i) find a gradient basis, and then (ii) generate the corresponding regions as explained in Sect. 2.3. Next section shows how to do the first step.

### 3.1 Computing a Basis of Gradient Vectors

We extend the concept of gradient of an event to sequences of events:

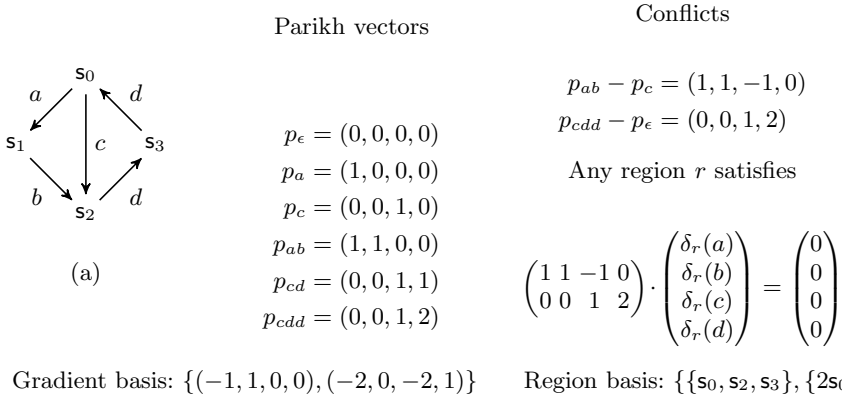
**Definition 13 (Gradient of a sequence).** *Let  $\sigma$  be a sequence of events, and  $r$  a region. The gradient of  $\sigma$  in  $r$ , denoted  $\delta_r(\sigma)$ , is  $\sum_{e \in \Sigma} \#(\sigma, e) \cdot \delta_r(e)$ .*

The following property is crucial for the method developed in this section to compute a gradient basis:

*Property 1.* Any region  $r$  of a TS has gradients such that any two paths  $s \xrightarrow{\sigma} s'$  and  $s \xrightarrow{\sigma'} s'$  satisfy that  $\delta_r(\sigma) = \delta_r(\sigma')$ .

Property 1 is automatically satisfied if there is only a single path connecting any two states, or the only paths between two states fire exactly the same events the same number of times (but possibly in different order). That is, if a state has the same *Parikh vector* no matter the path used to reach it.

**Definition 14 (Parikh vector, Parikh vector table).** *Given a TS  $A = \langle S, \Sigma, T, s_0 \rangle$ , the Parikh vector of a sequence  $\sigma$  is a vector  $p_\sigma \in \mathbb{N}^{|\Sigma|}$  such that  $p_\sigma(e) = \#(\sigma, e)$ . The set of Parikh vectors of a state  $s$ , denoted  $P_s$ , contains the Parikh vectors of all sequences  $\sigma$  that start from  $s_0$  and end in  $s$ . If all states in  $S$  have a single Parikh vector, *i.e.*  $|P_s| = 1$ , the Parikh vector table of TS  $A$  is a table with  $|S|$  columns, in which each column contains the Parikh vector of one state in  $S$ .*



**Fig. 4.** Computing the gradient basis of a TS. The symbol  $\epsilon$  is used for the empty sequence.

If  $P_s$  contains more than one element, by Property  $\square$ , any feasible gradient of a region  $r$  must make these Parikh vectors compatible, *i.e.* the multiplicity in  $r$  of state  $s$  must be the same no matter the path taken to reach it. In particular, for any state  $s$  and any two sequences  $\sigma$  and  $\sigma'$  such that  $s_0 \xrightarrow{\sigma} s$ ,  $s_0 \xrightarrow{\sigma'} s$  and  $p_\sigma \neq p_{\sigma'}$ , it must be true that  $\delta_r(\sigma) - \delta_r(\sigma') = 0$ . This is that

$$\sum_{e \in \Sigma} (p_{\sigma'}(e) - p_\sigma(e)) \cdot \delta_r(e) = 0. \tag{1}$$

We can use Eqn.  $\square$  as a building block of an algorithm that computes a gradient basis. The algorithm comprises two phases: (1) Traverse the TS, computing the Parikh vectors assigned to each state and recording the conflicts (Algorithm  $\square$  below), and (2) From the list of Parikh vector conflicts, build a system of equations using Eqn.  $\square$  from which we can derive the gradient basis. Algorithm  $\square$  shows the exploration phase of the algorithm. The function returns a set  $C$  of Parikh vector differences. Following Eqn.  $\square$ , any feasible gradient of the system must satisfy all these differences. We can write such condition in matrix form as  $M \cdot \Delta(r)^T = 0$ , where  $\Delta(r)^T$  is the gradient vector written as a column vector, and each row of matrix  $M$  contains one element of  $C$ .

*Example 3.* Consider the TS of Fig.  $\square$ . Starting from  $s_0$ , Parikh vectors of each state are computed. In some cases there are states that have different Parikh vectors assigned ( $s_2$  and  $s_0$ ). Such cases are recorded as conflicts, and the difference in the Parikh vectors is used to construct the matrix that enforces the equality of all conflicting Parikh vectors.

It is important to realize that in general (e.g. for cyclic TSs),  $P_s$  might be infinite but only a finite subset is needed by our algorithm: consider that in the example above the algorithm uses the conflict between  $p_{abdd}$  and  $p_\epsilon$  instead of the one between  $p_{cdd}$  and  $p_\epsilon$ . The result would be the same since  $\delta_r(ab) = \delta_r(c)$  once the conflict between  $p_{ab}$  and  $p_c$  is solved.

---

**Algorithm 1.** find Parikh vector conflicts

---

```

function FIND_CONFLICTS(TS  $A = \langle S, \Sigma, T, s_0 \rangle$ )
   $P_{s_0} \leftarrow \{(0, 0, \dots, 0)\}$  ▷ Assign zero Parikh vector
  for all  $s \in S - \{s_0\}$  do  $P_s \leftarrow \emptyset$  ▷ Initialize the rest
   $E \leftarrow \{s_0\}$  ▷ Set of states to explore
   $V \leftarrow \emptyset$  ▷ Set of visited states (whose arcs have been visited)
  while  $E \neq \emptyset$  do
    select  $s$  in  $E$ 
     $E \leftarrow E - \{s\}$  ▷ Remove  $s$  from the set of states to explore
    select  $p$  in  $P_s$ 
    for all  $s \xrightarrow{e} s' \in T$  do
       $p' \leftarrow p$ 
       $p'(e) \leftarrow p'(e) + 1$  ▷  $p'$  is one of the Parikh vectors of  $s'$ 
       $P_{s'} \leftarrow P_{s'} \cup \{p'\}$  ▷ Update set of Parikh vectors
      if  $s' \notin V$  then  $E \leftarrow E \cup \{s'\}$ 
    end for
     $V \leftarrow V \cup \{s\}$  ▷ Mark  $s$  as visited
  end while
   $C \leftarrow \emptyset$  ▷ Initialize set of conflicts
  for all  $s$  such that  $|P_s| > 1$  do
    select  $p$  in  $P_s$ 
    for all  $p'$  in  $P_s - \{p\}$  do  $C \leftarrow C \cup \{p - p'\}$ 
  end for
  return  $C$ 
end function

```

---

**Proposition 3.** *If  $M \cdot \Delta(r)^T = \mathbf{0}$ , then  $\Delta(r)$  is the gradient of a region.*

*Proof.* Let  $\sigma$  and  $\sigma'$  be two sequences  $s \xrightarrow{\sigma} s'$  and  $s \xrightarrow{\sigma'} s'$ . If  $r$  is a region, then  $\delta_r(\sigma) = \delta_r(\sigma')$ . Rewriting this expression we obtain  $(p_\sigma - p_{\sigma'}) \cdot \Delta(r)^T = \mathbf{0}$ . If both sequences have a common prefix/suffix, such that  $\sigma = \alpha\omega\beta$  and  $\sigma' = \alpha\omega'\beta$ , then  $p_\sigma - p_{\sigma'} = p_\omega - p_{\omega'}$ , so without loss of generality we can assume  $\sigma$  and  $\sigma'$  have no state in common besides  $s$  and  $s'$ . Let  $\gamma$  be a sequence without cycles such that  $s_0 \xrightarrow{\gamma} s$ . If  $\gamma$  is unique, then  $|P_s| = 1$ . Let  $p_s \in P_s$ . Now  $(p_\sigma - p_{\sigma'}) \cdot \Delta(r)^T = \mathbf{0}$  is equivalent to  $((p_s + p_\sigma) - (p_s + p_{\sigma'})) \cdot \Delta(r)^T = \mathbf{0}$ , thus  $((p_{\gamma\sigma}) - (p_{\gamma\sigma'})) \cdot \Delta(r)^T = \mathbf{0}$ , which is an equation in  $M \cdot \Delta(r)^T = \mathbf{0}$  if  $p_{\gamma\sigma} - p_{\gamma\sigma'}$  is not already 0. On the other hand, if  $\gamma$  is not unique and another  $s_0 \xrightarrow{\gamma'} s$  exists, then it is either possible that Algorithm 1 adds  $((p_{\gamma\sigma}) - (p_{\gamma\sigma'})) \cdot \Delta(r)^T = \mathbf{0}$ , which has been already considered, or  $((p_{\gamma\sigma}) - (p_{\gamma'\sigma'})) \cdot \Delta(r)^T = \mathbf{0}$  (or any other combination of the sequences). In such case, if  $p_\gamma = p_{\gamma'}$ , we are done. Otherwise, there is an equation guarantying  $p_\gamma = p_{\gamma'}$  in  $M$ , since we have a conflict. The induction is possible since the  $\gamma$  sequences are always decreasing in size. □

So the problem reduces to finding the solutions to the homogeneous linear system  $M \cdot \Delta(r)^T = \mathbf{0}$ . Each solution of this equation system identifies a feasible gradient in the TS. Note that the system requires to have solutions in the integer domain because, by definition, all gradients have to be integers.

Homogeneous linear systems have one trivial solution (*i.e.*  $\mathbf{0}$ ) and infinite non-trivial solutions. If the homogeneous linear system is represented by a matrix  $M$ , it is said that all these solutions form the *nullspace* of  $M$ . The nullspace of a matrix has a basis of solutions, that is, every solution to the homogeneous linear system can be obtained by linear combination of the solution vectors in the basis. Formally, if the basis of the nullspace of  $M$  is formed by the vectors  $\{\mathbf{y}_1, \mathbf{y}_2, \dots\}$ , then any solution  $\mathbf{x}$  can be written as a unique linear combination  $\mathbf{x} = \sum_i \lambda_i \mathbf{y}_i$ , with  $\lambda_i \in \mathbb{Q}$ . Consequently any integer basis of the nullspace of matrix  $M$  is a valid gradient basis, since any valid gradient can be written as a linear (rational) combination of these gradients.

There are several well-known methods to obtain a basis for the nullspace of a matrix [16]. Basically they involve performing a Gaussian elimination on matrix  $M$ . Since matrix  $M$  has  $|C|$  rows and  $|\Sigma|$  columns, the cost of such operation is  $O(|C|^2 \cdot |\Sigma|)$ . Once the basis has been computed, the only additional step to perform is to check if some of the resulting vectors contains a non-integer number. In such case, since all numbers are rational, the vector is multiplied by the minimum common multiple of all denominators to obtain an integer gradient.

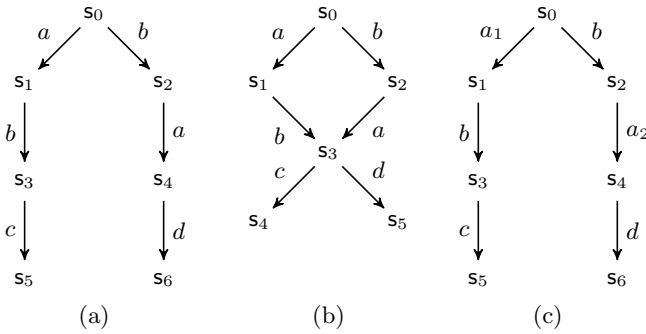
In the example of Fig. 4 the basis of the nullspace of matrix  $M$  is formed by gradient vectors  $(-1, 1, 0, 0)$  and  $(-2, 0, -2, 1)$ , from which we can obtain (using the technique shown in Sect. 2.3) the regions  $\{s_0, s_2, s_3\}$  and  $\{2s_0, s_3\}$ , respectively, which form a region basis.

The procedure presented in this section allows finding a region basis for any arbitrary TS by generating first a gradient basis. In some cases, however, this intermediate step can be avoided, as we will see in the next section.

### 3.2 Region Basis from a Language

In the area of Process Mining [17], the input object is typically a set of traces, *i.e.* a *language*, rather than a TS. In [7] three conversions from a language to a TS were proposed. The main difference between the conversions, namely *sequence*, *multiset* and *set*, is how they decide if the occurrence of an event in a trace produces a new state in the TS or just introduces an arc to an existing state. In this paper we will focus on the first two conversions. In the *sequence* conversion, two traces lead to the same state if they fire the same events in exactly the same order. For instance if  $L = \{abc, bad\}$ , the TS obtained from this conversion is shown in Fig. 5(a). In the *multiset* conversion events do not have to happen in the same order, but still it is required that the number of occurrences of each event to be equal. With the previous log, this yields the TS of Fig. 5(b).

Given a language, for the sequence and multiset conversions it is guaranteed that all paths leading to a state will have the same Parikh vector. Then, when computing the gradient basis there will be no conflict and we can choose an arbitrary set of  $|\Sigma|$  linearly independent gradients as a gradient basis. A simple option is to choose the *standard basis*, that is the basis formed by all the linearly independent vectors in which only one event has gradient one, and the rest have gradient zero (see an example below). Computing the corresponding regions is very natural when the standard basis is used, since we only need the Parikh



**Fig. 5.** (a) A TS. (b) Its quotient TS. The quotient TS accepts more traces, but their PN *without label splitting* is the same. (c) If label splitting is performed (for synthesis purposes), then the TS and its quotient TS coincide.

vector of each state, which has been already computed while searching for Parikh vector conflicts.

*Example.* The TS of Fig. 3(b) contains no Parikh vector conflict. Thus we will use the regions with gradients  $(1, 0, 0)$ ,  $(0, 1, 0)$  and  $(0, 0, 1)$ , *i.e.* the standard basis, as the basis. To compute their regions we use the Parikh vector in each state, shown in the *Parikh vector table* to the right. Each row corresponds to one of the regions, so that region with gradient  $(1, 0, 0)$  is  $\{s_1, s_3\}$ , region with gradient  $(0, 1, 0)$  is  $\{s_2, s_4\}$ , and gradient  $(0, 0, 1)$  belongs to region  $\{s_3, s_4\}$ . Notice that this is a different basis (analogously valid) than the one shown in Example 2.

Event	$s_0$	$s_1$	$s_2$	$s_3$	$s_4$
a	0	1	0	1	0
b	0	0	1	0	1
c	0	0	0	1	1

**Proposition 4.** *In a TS without Parikh vector conflicts, a row in the Parikh vector table corresponds to a region.*

*Proof.* For an event  $e$ , the value of the Parikh vector table for the row assigned to event  $e$  and the column of state  $s$  is  $p_s(e) = \#(\sigma, e)$ , being  $\sigma$  the unique path from the initial state to  $s$  in the TS, thus the multiplicity of the corresponding region  $r$  is  $r(s) = \#(\sigma, e)$ . We prove that  $r$  is a region by proving that the gradient of all events is constant. First of all, the value assigned to a state  $s$  is the same no matter which sequence  $\sigma$  is used to reach  $s$ , because the TS has no Parikh vector conflicts. For an event  $a \neq e$  the gradient is 0, since any arc  $s \xrightarrow{a} s'$  has a gradient  $r(s') - r(s) = 0$  as, if  $\sigma$  leads to  $s$ , then  $\#(\sigma a, e) = \#(\sigma, e)$ . Similarly, event  $e$  has gradient 1, because  $\#(\sigma e, e) = \#(\sigma, e) + 1$ .  $\square$

### Regions for sequential and multiset language representations

There can be significant differences when using either sequential or multiset conversions, since typically the sequential conversion yields TSs with much more states. So it is a relevant question to decide whether there is some advantage of the sequential conversion over the multiset conversion. As we will prove, as far as regions are concerned, there is no difference between both approaches.

**Definition 15.** States  $s$  and  $s'$  in a TS are said to be equivalent,  $s \equiv s'$ , if, for all region  $r$  of the TS,  $r(s) = r(s')$ . We denote the equivalence class of  $s$  as  $[s]$ .

**Proposition 5.** States  $s$  and  $s'$  in a TS  $A$  are equivalent if, for all region  $r$  in the region basis of  $A$ ,  $r(s) = r(s')$ .

The state equivalence relation partitions the set of states in equivalence classes. The TS that abstracts the behavior of a given TS at the level of the equivalence classes is called the *quotient TS*.

**Definition 16 (Quotient TS).** Let  $A = \langle S, \Sigma, T, s_0 \rangle$  be a TS. The quotient TS of  $A$ , denoted  $A_{/\equiv}$ , is a TS  $\langle S_{/\equiv}, \Sigma, T_{/\equiv}, [s_0] \rangle$ , where  $S_{/\equiv} = \{[s] \mid s \in S\}$  and  $T_{/\equiv} = \{[s] \xrightarrow{e} [s'] \mid s \xrightarrow{e} s' \in T\}$ .

Let us construct the quotient TS of Fig. 5(a). To determine which states are equivalent, we use Proposition 5 on a basis, which can be obtained by the method shown in the previous section. Since the TS is acyclic and has no conflicts, by Proposition 4, each row of the Parikh vector table below corresponds to one of the regions in the standard basis.

Using Proposition 5 any two states that have the same multiplicity in all the regions of the basis must have the same columns in the table. The only two states fulfilling this condition are  $s_3$  and  $s_4$ , which can be merged obtaining the TS shown in Fig. 5(b).

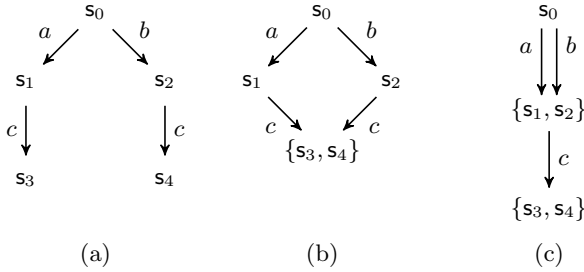
Event	$s_0$	$s_1$	$s_2$	$s_3$	$s_4$	$s_5$	$s_6$
a	0	1	0	1	1	1	1
b	0	0	1	1	1	1	1
c	0	0	0	0	0	1	0
d	0	0	0	0	0	0	1

**Theorem 2.** Let  $A = \langle S, \Sigma, T, s_0 \rangle$  be a TS, and  $A_{/\equiv} = \langle S_{/\equiv}, \Sigma, T_{/\equiv}, [s_0] \rangle$  be the quotient TS. For any canonical region  $r$  of  $A$  there is a canonical region  $r'$  in  $A_{/\equiv}$  such that  $\Delta(r) = \Delta(r')$  and  $r(s_0) = r'(s_0)$ , and vice versa.

*Proof.* Consider region  $r$  from  $A$ . Since it is a region we have that  $\forall s \xrightarrow{e} s' \in T, r(s') - r(s) = \delta_r(e)$ . And because it is normalized we have that  $\exists s : r(s) = 0$  and  $\forall s r(s) \geq 0$ . Consider the multiset  $r'$  such that  $r'([s]) = r(s)$ , we will prove that it is a canonical region with the same gradient and multiplicity in the initial state as  $r$ . Consider an arc  $[s] \xrightarrow{e} [s']$  in  $T_{/\equiv}$ , clearly  $r'([s']) - r'([s]) = r(s') - r(s) = \delta_r(e)$ . Thus all gradients are constant and  $r'$  is a region. Moreover, since all equivalent states  $s \in [s]$  have the same multiplicity in any region, then  $\exists [s] : r([s]) = 0$  and  $\forall [s] r([s]) \geq 0$ , which proves that  $r'$  is a canonical region. The proof in the other direction follows the same reasoning.  $\square$

**Proposition 6.** Let  $A_s$  and  $A_m$  be two TSs obtained from language  $L$  using the sequence conversion and the multiset conversion, respectively. Then  $A_m = A_{s/\equiv}$ .

*Proof.* We will prove that all the states of  $A_s$  that are equivalent are the ones that fire the same multiset of events. Consider the standard region basis for  $A_s$  and build its Parikh vector table. By definition two states are equivalent if they have the same multiplicity for all the regions in the basis. That is, if they have the same columns in the Parikh vector table, i.e. the two states have the same Parikh vector. As the Parikh vector is a representation of the multiset of events fired to reach the state, both states will be the same in  $A_m$ .  $\square$



**Fig. 6.** (a) A TS. (b) single sink version of (a), (c) merge of states  $s_1$  and  $s_2$  is possible since they are equivalent

A consequence of Proposition 6 and Theorem 2 is that multiset conversion provides the same information, in terms of regions, as the sequence conversion. This is relevant because the performance of some tools is specially affected by the number of states in the TS. This result holds even if the quotient TS contains more traces than the original TS. For instance, sequence  $abd$  is possible in Fig. 5(b), but not in Fig. 5(a). However Theorem 2 shows that the derived PN's are the same. Note that if label splitting 6 is allowed, then no extra behavior might be accepted by the quotient TS, as shown in (c).

**Beyond the multiset language representation**

In the previous section we have seen conditions allowing to reduce the number of states of a TS while obtaining the same net. This section proposes a more powerful reduction technique that considerably diminishes the size of the TS at the cost of forbidding some specific regions. The technique, named *common final marking* (CFM) reduction, has two steps:

- From a TS  $A$  obtained by multiset conversion, create a TS  $A'$  by merging all *sink states* (states without outgoing arcs) into a single state. We say that  $A'$  is the *single sink version* of  $A$ .
- Merge equivalent states in  $A'$ , by merging all states that are either reachable from a state  $s$  through the same event or reach the same state through a common event, until no further state is mergeable.

**Theorem 3.** *Let  $A$  be a TS obtained from a language and  $A'$  its single sink version. Consider a TS  $A''$  obtained from  $A'$  by merging all states that are either reachable from a state  $s$  through the same event or reach the same state through a common event, until no further state is mergeable. Let  $N'$  and  $N''$  be the PN's including all the regions of  $A'$  and  $A''$ , respectively. Then,  $\mathcal{L}(N') \supseteq \mathcal{L}(A)$  and  $N' = N''$ .*

*Proof.* Merging the sink states of  $A$  does not introduce any new trace, so  $\mathcal{L}(A') = \mathcal{L}(A)$ . However,  $A$  has no conflict while  $A'$  can contain some of them. This yields a smaller region basis, thus some regions of  $A$  are no longer feasible in  $A'$ ,

consequently PN  $N'$  obtained from  $A'$  satisfies  $\mathcal{L}(N') \supseteq \mathcal{L}(A)$ . Now consider two states  $s_1$  and  $s_2$  such that the transitions  $s \xrightarrow{e} s_1$  and  $s \xrightarrow{e} s_2$  exist in  $A'$ . In any possible region  $r$ , both states will have the same multiplicity since  $r(s_1) = r(s) + \delta_r(e) = r(s_2)$ . Thus, by Proposition 5,  $s_1 \equiv s_2$ , and both states can be merged. The same applies if  $s_1 \xrightarrow{e} s$  and  $s_2 \xrightarrow{e} s$ . Since  $A''$  is simply  $A'$  but merging equivalent states, by Theorem 2,  $N''$  and  $N'$  must be the same.  $\square$

For instance in Fig. 6 we can see a TS (from Fig. 3), that could be derived from  $L = \{ac, bc\}$ . Both the sequential and the multiset conversion yield the same TS, shown in (a). This TS has two sink states  $s_3$  and  $s_4$ , which can be merged obtaining a TS, depicted in (b), with the same language. This merged state has two incoming arcs with the same label, thus, the predecessors of such arcs, namely  $s_1$  and  $s_2$  can be also safely merged, since they will be assigned the same multiplicity in every possible region.

### 4 Generating a PN from a Basis

Once the region basis is available, we can generate a PN from it. A naive strategy would be to use a brute-force approach and generate some amount of regions, and then remove the redundant ones among them using, for instance, the techniques in 4. However this approach is clearly inefficient.

An alternative generation scheme is to try to find the minimal canonical regions. The straightforward approach would be to have a set of the currently minimal regions found so far in the exploration, and every time a new region is generated, check against all the regions in the set whether it is minimal or not. However, this method requires to perform many subregion checks per region, and most of the times the regions checked are not minimal.

Our proposal (Algorithm 2), denoted *minimal canonical region search*, prevents from checking the minimality of regions that are guaranteed not to be minimal. If  $B = \{r_1, \dots, r_n\}$  is the region basis, the algorithm computes the set of minimal canonical regions  $R$  in a DFS manner. It starts with the empty region  $\emptyset$  to whom normalized basis regions  $\downarrow(c_i \cdot r_i)$ , with  $c_i \neq 0$ , are added. The first addition creates a normalized region  $r = \downarrow(c_i \cdot r_i)$  from a single basis region. These type of regions are always checked for minimality since *size* is always 1 in such case. Thus, they are added into  $R$  if no smaller region is present in  $R$  (line 14). This guarantees that  $R$  contains either  $r$  or one of its subregions. From that point on, combinations including  $r$  are explored, by adding other normalized basis regions (line 19). Let  $r' = \downarrow r + \downarrow(c_j \cdot r_j)$  be one of such explored regions. It is trivially true that  $r' \supseteq \downarrow r$ . Since  $\downarrow r$  is a normalized region, if  $r'$  is also normalized, it follows that  $\downarrow r' \supseteq \downarrow r$ , thus  $r' \supseteq r$  and  $r'$  is not minimal.

Consequently, the algorithm is devised to detect whether the addition of some region basis  $(c_i \cdot r_i)$  to a region  $r$  produces a non-normalized region. This check is performed in line 3, based on the fact that, if  $r$  is already normalized, then the multiplicity of any state must be the same in  $r$  or  $\downarrow r$ . In line 3 normalization of  $r$  is tested in the initial state  $s_0$  with the condition  $\downarrow r(s_0) \neq r(s_0)$ . Only the



regions satisfying this condition are possible minimal canonical regions, and are checked against all the regions in the current set  $R$ .

---

**Algorithm 2.** `mcr_search`


---

```

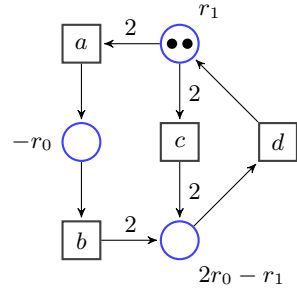
1: procedure MCR_RECURSIVE( $r, size, pos$ )
2:    $nr \leftarrow \downarrow r$  ▷ Normalize  $r$ 
3:   if  $size = 1 \vee nr(s_0) \neq r(s_0)$  then ▷ Check  $r = c_i \cdot r_i$ , or  $r$  non-normalized
4:     if  $\exists e : \delta_r(e) < 0$  then ▷ Regions with positive gradients are useless
5:        $useful \leftarrow true$  ▷ Initially consider  $nr$  is minimal
6:       for all  $mr \in R$  do
7:         if  $mr \subseteq nr$  then
8:            $useful \leftarrow false$  ▷ If  $nr$  is not minimal discard it
9:           break
10:        else
11:          if  $nr \subseteq mr$  then  $R \leftarrow R - \{mr\}$  ▷ Remove  $mr$  (not minimal)
12:          end if
13:        end for
14:        if  $useful$  then  $R \leftarrow R \cup \{nr\}$  ▷ Add  $nr$  as minimal region
15:        end if
16:      end if
17:      if  $size < agg$  then ▷ Check aggregation factor
18:        for all  $i \in [pos, |B|]$  and  $j \in [minval, maxval] - \{0\}$  do
19:           $mcr\_recursive(nr + \downarrow(j \cdot r_i), size + 1, pos + 1)$ 
20:        end for
21:      end if
22:    end procedure
23:
24: function MCR_SEARCH
25:    $R \leftarrow \emptyset$  ▷ Set of minimal canonical regions
26:    $mcr\_recursive(\emptyset, 0, 1)$  ▷ Call recursive function
27:   return  $R$ 
28: end function

```

---

The algorithm uses the following global variables:  $R$  is the set of minimal canonical regions encountered so far,  $B$  is the region basis found by the methods described in Sect. 3,  $agg$  is an aggregation factor that bounds the number of different basis regions that can be used to obtain a new region. Finally,  $minval \leq -1$  and  $maxval \geq 1$  bound the number of times that a basis region can appear in a combination. The last three variables are user defined parameters, that allow the user to control the amount of exploration performed in the region space that takes place in lines 17 to 19 of the algorithm. Formally, only regions  $r$  are explored such that, for all event  $e$ ,  $\delta_r(e) = \sum c_i \delta_{r_i}(e)$ , where  $minval \leq c_i \leq maxval$  and  $|\{c_i \mid c_i \neq 0\}| \leq agg$ . Note that if the region basis comes from the standard gradient basis, then  $-minval$  is the maximum allowed value for the weight of an incoming arc to a transition,  $maxval$  is the maximum allowed value for the weight of an outgoing arc of a transition and  $agg$  is the maximum number of arcs that a place can have.

$$\begin{aligned}
 r_0 &= \{s_0, s_2, s_3\} & \downarrow(-r_0) &= \{s_1\} \\
 r_1 &= \{2s_0, s_3\} & \downarrow(-r_1) &= \{2s_1, 2s_2, s_3\} \\
 \\ 
 \downarrow(-r_0) + \downarrow(-r_1) &= \{3s_1, 2s_2, s_3\} \supseteq \downarrow(-r_0) \\
 \downarrow(-r_0) + r_1 &= \{2s_0, s_1, s_3\} \supseteq \downarrow(-r_0) \\
 \downarrow(-r_0) + 2r_1 &= \{2s_0, 2s_1, s_3\} \supseteq \downarrow(-r_0) \\
 r_0 + \downarrow(-r_1) &= \{s_0, 2s_1, 3s_2, 2s_3\} \neq \downarrow(r_0 + \downarrow(-r_1)) \\
 \downarrow(r_0 + \downarrow(-r_1)) &= \{s_1, 2s_2, s_3\} \supseteq \downarrow(-r_0) \\
 2r_0 + \downarrow(-r_1) &= \{2s_0, 2s_1, 4s_2, 3s_3\} \neq \downarrow(2r_0 + \downarrow(-r_1)) \\
 \downarrow(2r_0 + \downarrow(-r_1)) &= \{2s_2, s_3\} \not\supseteq \downarrow(-r_0) \Rightarrow \text{added to } R
 \end{aligned}$$



**Fig. 7.** Some of the regions explored by the algorithm mining the TS in Fig. 4 (only the shadowed regions are checked against regions in  $R$ ), and the final PN obtained

Although the algorithm does not produce  $k$ -bounded nets, it is not difficult to adapt it to fulfill such requirement. Similarly, the output of the algorithm are pure PNs, however general PNs can be easily generated from the latter as shown in [15].

To illustrate the behavior of the algorithm, we will follow the first steps of the PN generation using the region basis obtained in Fig. 4. To exemplify the impact of some of the parameters, we will use  $minval = -1$  and  $maxval = 2$ , without limiting the possible combinations of the regions in the basis (*i.e.* using the size of the basis, in this case 2, as the aggregation factor  $agg$ ). Some of the regions explored and the final PN can be seen in Fig. 7. The values of the  $minval$  and  $maxval$  parameters are loosely related to the weights in the arcs one would expect in the set of minimal regions. In this example, using  $maxval = 1$  would prevent the algorithm from finding one of the places in the net.

Let us name  $r_0$  and  $r_1$  the two region basis in the example, namely  $\{s_0, s_2, s_3\}$  and  $\{2s_0, s_3\}$ . Since  $r_0$  and  $r_1$  are already normalized and to ease the notation, we will simply write  $k \cdot r_0$ , when  $k$  is a positive scalar, instead of  $\downarrow(k \cdot r_0)$ , since these regions are normalized too. First region explored is  $\downarrow(-r_0)$ , which is added into the  $R$  set of minimal canonical regions, because all regions formed using a single region basis are always tested against the regions in  $R$  (variable  $size$  is always 1 in such cases, and the condition in line 3 evaluates to true) and  $R$  is initially empty. After that, regions  $\downarrow(-r_0) + \downarrow(-r_1)$ ,  $\downarrow(-r_0) + r_1$  and  $\downarrow(-r_0) + 2r_1$  are explored. None of them is non-normalized (see Fig. 7), thus are discarded without been checked for minimality.

Next  $r_0$  is checked, and goes into the  $R$  list since it is obviously not a superset of  $\downarrow(-r_0)$ . From the following combinations  $r_0 + \downarrow(-r_1)$ ,  $r_0 + r_1$  and  $r_0 + 2r_1$ , only the first one corresponds to a non-normalized region. However when checked against the regions in  $R$ , it turns out that it is a superregion of  $\downarrow(-r_0)$ , thus it is not included in the list. The  $2r_0$  region (not shown in the figure) has the trivial subregion  $r_0$ , but one of its descendants,  $2r_0 + \downarrow(-r_1)$  is minimal. Finally,

regions  $\downarrow(-r_1)$ ,  $r_1$  and  $2r_1$  are checked and discarded with the exception of  $r_1$ . The exploration concludes after generating 15 regions with a list of four minimal canonical regions. From this set, using the simplification techniques described in [4], one of them is removed, yielding the PN shown in Fig. 7.

**Proposition 7.** *Given a TS  $A$ , if a basis of its regions is used in Algorithm 2, then the set of regions returned by the algorithm correspond to a PN  $N$  such that  $\mathcal{L}(N) \supseteq \mathcal{L}(A)$ .*

## 5 Experiments

All the results were obtained on a PC with an Intel Core Duo at 2.10Ghz and 2Gb of RAM, running the 2.6 Linux kernel. For the experiments we limited the amount of memory and time that could be used by the tools to 1Gb and 10000 seconds respectively. Table 1 shows some relevant information of the logs used in the experiments. For each benchmark we give the number of traces it contains ( $\#cases$ ), the number of different events present in the log ( $|\Sigma|$ ), the number of states of the corresponding execution tree obtained by the sequential conversion ( $|S_s|$ ), the number of states of the TS obtained by the multiset conversion ( $|S_m|$ ), and the number of states after CFM reduction ( $|S_c|$ ). The time required to build the TS by each type of conversion is given in columns  $T_s$ ,  $T_m$  and  $T_c$ , respectively. Column  $|C|$  indicates the number of conflicts present in the TS obtained by CFM reduction, while  $|B|$  is the size of the corresponding region basis.

**Table 1.** Logs from [9] used in the experiments

Log	$\#cases$	$ \Sigma $	$ S_s $	$ S_m $	$ S_c $	$T_s$	$T_m$	$T_c$	$ C $	$ B $
a12f0n00_1	200	12	25	18	13	0	0	0	2	10
a12f0n00_5	1800	12	25	18	13	0	0	0	2	10
a22f0n00_1	100	22	1309	751	86	0	0	0.1	10	16
a22f0n00_5	900	22	9867	3291	80	0.1	0.1	0.3	6	16
a32f0n00_1	100	32	2011	1378	614	0	0.1	0.1	28	26
a32f0n00_5	900	32	16921	5544	481	0.1	0.3	0.4	10	26
t32f0n00_1	200	33	7717	7167	5846	0.1	0.4	0.4	119	27
t32f0n00_5	1800	33	64829	50436	2870	0.4	3.6	4.8	35	27
a42f0n00_1	100	42	2865	2568	1864	0	0.1	0.2	74	35
a42f0n00_5	900	42	24366	15816	8221	0.1	1.1	1.1	192	35

As expected the number of states produced by the multiset conversion is inferior to the sequential conversion, specially when combined with the CFM reduction. Since savings can be dramatic in some cases, like the `t32f0n00_5` log, we have used the TSs obtained by the multiset conversion with CFM reduction.

We compare the performance and the quality of three tools: the `Parikh` miner in the `ProM` tool, `genet` and `rbminer`. The `Parikh` miner [9] uses the language-based theory of regions combined with ILP, `genet` implements the classical TS-based approach with a symbolic representation of the TSs, and the `rbminer`

**Table 2.** Mining of large logs

Log	genet			Parikh			rbminer		
	P/F	Time	App.	P/F	Time	App.	P/F	Time	App.
a12f0n00_1	11/25	0.1	1.0	11/25	1	1.0	11/25	0.1	1.0
a12f0n00_5	11/25	0.1	1.0	11/25	0.7	1.0	11/25	0.1	1.0
a22f0n00_1	19/49	0.3	0.95	19/49	3	0.95	19/49	0.1	0.93
a22f0n00_5	19/49	0.3	0.94	19/49	23	0.95	19/49	0.1	0.94
a32f0n00_1	32/75	718	0.94	31/73	25	0.93	32/75	2	0.94
a32f0n00_5	31/73	1	0.95	31/73	112	0.93	31/73	2	0.95
t32f0n00_1	memout			30/72	288	0.99	31/74	8	0.92
t32f0n00_5	memout			30/72	9208	0.99	30/72	5	0.92
a42f0n00_1	memout			44/109	154	1.0	52/131	10	1.0
a42f0n00_5	timeout			44/101	1557	1.0	46/107	33	1.0

tool implements the methodology described in this paper. For each method we provide the number of places and arcs (column  $P/F$ ) of the mined PN (the number of transitions coincides with  $|\Sigma|$  since no label splitting is performed), the time in seconds to obtain the PN from the TS, and the well-known quality measure called *appropriateness* [18]. This metric quantifies to which extent the model describes the observed behavior, combined with the clarity degree of the model. It is normalized to be a real number between 0 (low) and 1 (high). All the benchmarks were mined using an aggregation factor of 4 for **rbminer**, with  $minval = -1$ ,  $maxval = 1$  and  $k = 1$ .

The benefits of using basis of regions are twofold. First, the memory consumption is very low: in all the experiments the maximum amount of memory used by **rbminer** was 10Mb. This is a clear advantage over other approaches, notably **genet**, which is very memory demanding. Second, the running times are, in general, much lower. The crucial step in obtaining such improvements is the use of the CFM reduction [2]. In terms of quality the results are quite similar across all tools. Note that in some cases PNs with the same number of places and arcs have different appropriateness because they are not identical.

In addition to the experiments on mining logs, which always yield acyclic TSs, we have conducted a number of additional experiments on cyclic TSs [3], that were obtained by computing the RG of several PNs. A comparison of the mining capabilities of the **genet** and **rbminer** tools [4] shows that, in both cases, no extra behavior was included in the derived PN (for the cases in which **genet** could complete). However the resulting PNs were very different in terms of compactness. In all cases **rbminer** could reconstruct the original PNs from which the TSs were derived. The aggregation factor was set in each case to the value where synthesis was achieved. Note that for some benchmarks the values are quite low, showing that in many cases a very shallow partial exploration of the region space is enough to obtain remarkable results.

<sup>2</sup> We have repeated the experiments using the multiset conversion alone, and the running times became similar to the ones obtained by the **Parikh** tool.

<sup>3</sup> A description of these benchmarks can be found in [11].

<sup>4</sup> The Parikh miner cannot handle cyclic TSs, thus it does not appear in Table 3.

**Table 3.** Mining of cyclic TSs

Bench.	S	Σ	C	B	genet		rbminer	
					P/F	Time	Agg	P/F Time
PC(8,3)	1024	17	8	9	27/86	2	9	18/50 0.1
PC(8,5)	1536	17	8	9	42/158	83	9	18/50 0.1
PC(9,6)	3584	19	9	10	62/256	332	10	20/56 1
SR(6,4)	4077	24	6	18	89/490	20	6	25/60 32
SR(7,5)	16362	28	7	21	241/1865	1190	7	29/70 1565
BP(8)	6561	10	1	8	16/32	1320	2	16/32 0
BP(9)	19683	11	1	9	18/36	4561	2	18/36 0
BP(10)	59049	12	1	10	timeout		2	20/40 0.1

## 6 Related Work

The work presented has some relations with the theory developed in [14,15], being the contributions of this paper algorithms built on top of that theory.

Related approaches based on the language-based theory of regions are [49], which are based in the seminal work presented in [3]. Informally, these approaches build also a basis of regions (but only allowing positive combinations, thus finding a basis formed by all minimal canonical regions) by solving an homogeneous linear equation system that is proportional to the set of words (and prefixes of the words) that appear in the language, and some of them are also proportional with the set of *wrong continuations* of words, i.e. words not appearing in the language. This makes the language-based approach to suffer for large inputs, as it is demonstrated in the previous section. However, the TS that arises from a language can be greatly simplified (as explained in Sect. 3.2), which in turn alleviates drastically the size of the object needed for deriving a region basis, thus making the approach of paper a good candidate for large inputs.

## 7 Conclusions

This paper presents a fresh look at the problem of deriving a PN from a TS using the theory of regions. By combining ideas that has been applied in the language-based theory of regions (i.e. the generation of a region basis), together with drastic simplifications of the input TS, the approach handles inputs with better run-time and similar quality than current approaches for Process Mining. The theory has been implemented in the tool `rbminer` [10].

## Acknowledgements

This work has been supported by projects FORMALISM (TIN2007-66523) and TIN2007-63927.

## References

1. Desel, J., Reisig, W.: The synthesis problem of Petri nets. *Acta Inf.* 33(4), 297–315 (1996)
2. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures. Part I, II. *Acta Informatica* 27, 315–368 (1990)
3. Badouel, E., Bernardinello, L., Darondeau, P.: Polynomial algorithms for the synthesis of bounded nets. In: Mosses, P.D., Schwartzbach, M.L., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 364–383. Springer, Heidelberg (1995)
4. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007)
5. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers* 47(8), 859–882 (1998)
6. Carmona, J., Cortadella, J., Kishinevsky, M.: New region-based algorithms for deriving bounded Petri nets. *IEEE Transactions on Computers* 59(3) (2009)
7. van der Aalst, W., Rubin, V., Verbeek, H., van Dongen, B., Kindler, E., Günther, C.: Process mining: a two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling* (2009)
8. 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)
9. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 368–387. Springer, Heidelberg (2008)
10. Solé, M.: rbminer, <http://www.lsi.upc.edu/~jcarmona/rbminer/rbminer.html>
11. Solé, M., Carmona, J.: Process mining from a basis of state regions. Technical Report LSI-09-35-R, Software Dept., Universitat Politècnica de Catalunya (2009)
12. Murata, T.: Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, 541–580 (April 1989)
13. Mukund, M.: Petri nets and step transition systems. *Int. Journal of Foundations of Computer Science* 3(4), 443–478 (1992)
14. Bernardinello, L., Michelis, G.D., Petruni, K., Vigna, S.: On the synchronic structure of transition systems. In: Desel, J. (ed.) *Structures in Concurrency Theory*, Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), Berlin, May 11–13, pp. 69–84 (1995)
15. Badouel, E., Darondeau, P.: Theory of regions. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
16. Kalman, D.: Basic null space calculations. *The College Mathematics Journal* 15(1), 42–47 (1984)
17. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.* 16(9), 1128–1142 (2004)
18. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. *Inf. Syst.* 33(1), 64–95 (2008)

# Separability in Persistent Petri Nets

Eike Best<sup>1</sup> and Philippe Darondeau<sup>2</sup>

<sup>1</sup> Parallel Systems, Department of Computing Science  
Carl von Ossietzky Universität Oldenburg, D-26111 Oldenburg, Germany

`eike.best@informatik.uni-oldenburg.de`

<sup>2</sup> INRIA, Centre Rennes - Bretagne Atlantique

Campus de Beaulieu, F-35042 Rennes Cedex

`Philippe.Darondeau@inria.fr`

**Abstract.** We prove that plain, bounded, reversible and persistent Petri nets are weakly and strongly separable.

## 1 Introduction

Given a place/transition Petri net  $N = (N, M_0)$  with initial marking  $M_0$  and a number  $k \in \mathbb{N}$ , one may consider the  $k$ -multiple net  $k \cdot N = (N, k \cdot M_0)$ , where every place holds  $k$  times the number of tokens it holds in  $M_0$ . This paper investigates the relationship between  $N$  and  $k \cdot N$ . The net  $k \cdot N$  will be called *strongly separable* if every firing sequence starting at  $k \cdot M_0$  belongs to the shuffle product of  $k$  firing sequences starting at  $M_0$ , and *weakly separable* if the Parikh vector of every firing sequence starting at  $k \cdot M_0$  is the sum of the Parikh vectors of  $k$  firing sequences starting at  $M_0$ . Our notions of strong and weak separability were called serializability and separability, respectively, in [7] where they were first introduced (together with another notion also called weak separability but even weaker than ours). Strong separability was proved in [7] for state machines and acyclic marked graphs (with the workflow property). Weak separability was proved in [2] for marked graphs, a strict subclass of persistent nets. In this paper, we prove both weak and strong separability for plain, bounded, reversible and persistent nets (*pbrp-nets*, for short), thus settling a conjecture made in [1]. Boundedness means that the set of reachable markings is finite. Reversibility means that the initial marking is reachable from every other reachable marking. Persistency means that at any reachable marking, an enabled transition is never disabled by the firing of another transition.

The remaining sections of the paper are organized as follows. Section 2 presents the technical background. Section 3 establishes two lemmata showing the stability of pbrp nets  $k \cdot N$  under division by  $k$ . Section 4 establishes a crucial result stating that, if a pbrp net  $k \cdot N$  with  $k \geq 2$  has a single minimal realizable T-invariant  $X$ , then  $X \leq 1$ . Section 5 introduces the properties of weak and strong separability, which are shown to hold in sections 6 and 7, respectively, for pbrp nets  $k \cdot N$  with a single minimal realizable T-invariant. Both properties are extended to general pbrp nets  $k \cdot N$  in section 8. A supplementary result and some open questions are briefly mentioned in section 9.

## 2 Basic Definitions, and Earlier Results

We assume familiarity with the notation  $(P, T, F, M_0)$  for marked Petri nets.

### 2.1 Petri Nets, Boundedness, Reversibility, and Persistency

A net  $N = (P, T, F, M_0)$  is *plain* (or *ordinary*) if arc weights do not exceed 1 (i.e.,  $\text{cod}(F) \subseteq \{0, 1\}$ ).  $N$  is *m-bounded* if  $M(p) \leq m$  for every place  $p$  in every reachable marking  $M \in [M_0]$ , and *bounded* if it is *m-bounded* for some  $m \in \mathbb{N}$ .  $N$  is *persistent*, if whenever  $M[t_1]$  and  $M[t_2]$  for a marking  $M \in [M_0]$  and two transitions  $t_1 \neq t_2$ , then  $M[t_1 t_2]$ .  $N$  is *reversible* if  $M_0 \in [M]$  for every  $M \in [M_0]$ . In the sequel, plain, bounded, reversible and persistent Petri nets are called pbrp-nets. Figure 1 shows a pbrp-net and its reachability graph.

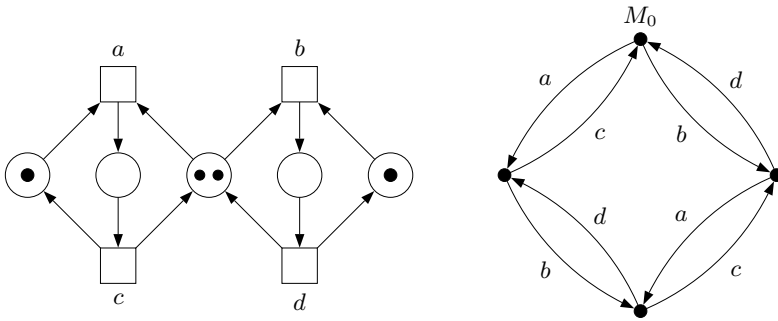


Fig. 1. A pbrp net (l.h.s.) and its reachability graph (r.h.s.)

### 2.2 Parikh Vectors, Permutation Equivalence, and Keller’s Theorem

For a finite sequence of transitions  $\sigma \in T^*$ , the *Parikh vector*  $\Psi(\sigma)$  of this sequence is a vector of natural numbers with index set  $T$ , where  $\Psi(\sigma)(t)$  is the number of occurrences of  $t$  in  $\sigma$ . The *marking equation* states that if  $M[\sigma]M'$ , then  $M' = M + C \cdot \Psi(\sigma)$ .

Two transition sequences  $\sigma \in T^*$  and  $\sigma' \in T^*$  are said to be *permutations of each other from marking M* (written  $\sigma \equiv_M \sigma'$ ) if they are both firable at  $M$  and they have the same Parikh vector.

By  $\tau \overset{\bullet}{\sigma}$ , we denote the *residue* of  $\tau$  left after cancelling successively in this sequence the leftmost occurrences of all symbols from  $\sigma$ , read from left to right. Formally,  $\tau \overset{\bullet}{\sigma}$  is defined by induction on the length of  $\sigma$ :  $\tau \overset{\bullet}{\varepsilon} = \tau$ ;  $\tau \overset{\bullet}{t} = \tau$ , if there is no label  $t$  in  $\tau$ , and the sequence obtained by erasing the leftmost  $t$  in  $\tau$ , otherwise; and  $\tau \overset{\bullet}{(t\sigma)} = (\tau \overset{\bullet}{t}) \overset{\bullet}{\sigma}$ .

Keller’s theorem [8] states that in a persistent net, if  $\tau$  and  $\sigma$  are two transition sequences firable at some reachable marking  $M \in [M_0]$ , then  $\tau(\sigma \overset{\bullet}{\tau})$  and  $\sigma(\tau \overset{\bullet}{\sigma})$  are also firable from  $M$ , and  $\tau(\sigma \overset{\bullet}{\tau}) \equiv_M \sigma(\tau \overset{\bullet}{\sigma})$ . Furthermore, the marking reached after  $\tau(\sigma \overset{\bullet}{\tau})$  equals the marking reached after  $\sigma(\tau \overset{\bullet}{\sigma})$ .



### 2.3 T-Invariants and Cycles

The *incidence matrix*  $C$  of a net  $(P, T, F)$  is a  $P \times T$ -matrix of integers where the entry corresponding to a place  $p$  and a transition  $t$  is, by definition, equal to the number  $F(t, p) - F(p, t)$ . A  $T$ -invariant  $J$  is a vector of integers with index set  $T$  satisfying  $C \cdot J = 0$ . When comparing vectors with scalars, such as here, we always mean this *componentwise*.  $J$  is called *semipositive* if  $J \geq 0$  and  $J$  is not the null vector. Throughout the paper, we will only consider semipositive T-invariants, and for succinctness, we will just call them “T-invariants”. Two (semipositive) T-invariants  $J$  and  $J'$  are called *transition-disjoint* if  $J \cdot J' = 0$ .

Two sequences  $\tau, \sigma \in T^*$  are called *Parikh-equivalent* if  $\Psi(\tau) = \Psi(\sigma)$ . In any Petri net,  $\sigma \equiv_M \tau$  entails  $\Psi(\sigma) = \Psi(\tau)$ , and  $\Psi(\sigma) = \Psi(\tau)$  entails also  $\sigma \equiv_M \tau$  if  $M$  is a reachable marking and both sequences  $\sigma$  and  $\tau$  are fireable at  $M$ .

Let  $M \in [M_0]$ . A sequence of transitions  $M[\tau]M$  is called a *cycle*. By the marking equation, for any cycle  $M[\sigma]M$ , the Parikh vector  $\Psi(\sigma)$  of this cycle is a T-invariant. A T-invariant is called *realizable* if it coincides with the Parikh vector of some cycle.

A cycle  $M[\tau]M$  is called *simple* if there is no permutation  $\tau' \equiv_M \tau$  such that  $\tau' = \tau_1\tau_2$ ,  $M[\tau_1]M$ ,  $M[\tau_2]M$ , and  $\tau_1 \neq \varepsilon \neq \tau_2$ . For example, in Figure 1(r.h.s.),  $M_0[ac]M_0$  is simple, but  $M_0[abcd]M_0$  is not simple, in view of the permutation  $M_0[ac]M_0[bd]M_0$ .

The following results from [3] will be used in the sequel.

**Theorem 1.** DECOMPOSING CYCLES OF REVERSIBLE PERSISTENT NETS

Let  $N = (P, T, F, M_0)$  be a bounded, reversible, and persistent Petri net. There exists a finite set  $\mathcal{B}$  of semipositive T-invariants such that they are transition-disjoint and every cycle  $M[\rho]M$  in the reachability graph of  $N$  can be decomposed, up to permutations, to some sequence  $M[\rho_1]M[\rho_2]M \dots [\rho_n]M$  of cycles with all Parikh vectors  $\Psi(\rho_i)$  in  $\mathcal{B}$ . Moreover,  $\mathcal{B}$  can be chosen as the set of Parikh vectors of simple cycles through any fixed state of  $N$ .

**Theorem 2.** DECOMPOSING REVERSIBLE PERSISTENT NETS

Let  $N = (P, T, F, M_0)$  be a bounded, reversible, and persistent net. Suppose that  $\mathcal{B} = \{X_1, \dots, X_n\}$ , thus at any reachable marking,  $N$  generates  $n$  simple cycles with transition disjoint Parikh vectors  $X_1, \dots, X_n$ . Then there are  $n$  bounded, persistent and reversible nets  $N_1, \dots, N_n$ , such that each net  $N_i$  has exactly one minimal realizable T-invariant  $X_i$  and the reachability graph of  $N$  is isomorphic to the reachability graph of the disjoint sum of the nets  $N_1, \dots, N_n$ .

‘Disjoint sum’ means that there is no place merging. The respective nets  $N_i$  constructed for  $i = 1, \dots, n$  in the proof of Theorem 2 are defined as  $N_i = (P, T_i, F_i, M_0)$  where  $T_i = \{t \in T \mid X_i(t) \neq 0\}$  and  $F_i$  is the induced restriction of  $F$  on  $(P \times T_i) \cup (T_i \times P)$ . In particular, all nets  $N_i$  have the same initial marking  $M_0$  as  $N$ . This remark is crucial to the use of Theorem 2 made later.

### 3 Multiples of a Net, Persistency, and the pbrp Properties

In this paper, we study  $k$ -multiples of nets as follows. Let  $N$  be a net and let  $k \geq 1$  be some positive integer number. For a marking  $M$ , the  $k$ -multiple marking  $k \cdot M$  is defined by  $(k \cdot M)(s) = k \cdot (M(s))$  for every place  $s$ . The net  $k \cdot N$  is the same as the net  $N$  except that the initial marking  $k \cdot M_0$  replaces the initial marking  $M_0$  of  $N$  (thus,  $1 \cdot N$  is the same as  $N$ ). The net  $k \cdot N$  is called a  $k$ -net, for short. An example is shown in Figure 2. A marking  $L$  which is of the form  $k \cdot M$ , that is, which assigns to every place a multiple of  $k$  as tokens, is called a  $k$ -marking.

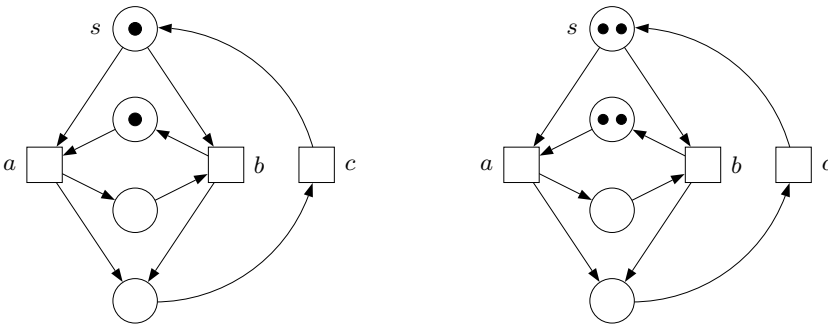


Fig. 2. A persistent Petri net (l.h.s.) and its 2-multiple (r.h.s.)

In this section, we show that the pbrp properties are preserved under scalar division of nets. Similar properties do not hold in general for multiplication. It is easy to construct a net  $N$  which is bounded, or persistent, or reversible while  $k \cdot N$  is not. For persistency, Figure 2 can be taken as a counterexample. Plainness is obviously preserved by division and will henceforth be assumed of all nets. The proof of the first lemma below is easy and omitted (but a proof is in [4], the full version of this paper).

**Lemma 1.** DIVISION PRESERVES BOUNDEDNESS AND PERSISTENCY

Let  $N$  be plain. Let  $k \geq 1$  and let  $k \cdot N$  be bounded (persistent). Then  $N$  is also bounded (respectively, persistent).

**Lemma 2.** DIVISION PRESERVES REVERSIBILITY

Let  $k \geq 1$  and let  $k \cdot N$  be pbrp. Then  $N$  is reversible.

**Proof:** Let  $M_0$  be the initial marking of  $N$  and suppose  $M_0[\alpha]M$ . As  $k \geq 1$ , also  $k \cdot M_0[\alpha]L$  in  $k \cdot N$  for the marking  $L = M + (k-1) \cdot M_0$ . Because  $k \cdot N$  is reversible,  $L[\beta]k \cdot M_0$  for some sequence  $\beta$ . Combining this with  $k \cdot M_0[\alpha]L$ , we get  $k \cdot M_0[\alpha\beta]k \cdot M_0$ .

Executing  $k$  times the cycle just found yields  $k \cdot M_0[(\alpha\beta)^k]k \cdot M_0$ . Let  $t_1$  be the first transition of  $(\alpha\beta)^k$ . Because  $k \cdot M_0[t_1]$  and the net is plain, also  $M_0[t_1]$ ,

say that  $M_0[t_1]M_1$ . Then also  $k \cdot M_0[t_1^k]$ , and of course,  $k \cdot M_0[t_1^k]k \cdot M_1$ . Keller's theorem applied in  $k \cdot N$  yields  $k \cdot M_0[t_1^k]k \cdot M_1[(\alpha\beta)^k \bullet t_1^k]k \cdot M_0$ . As  $(\alpha\beta)^k$  contains  $t_1$  a positive multiple of  $k$  times, the Parikh vector of the sequence  $(\alpha\beta)^k \bullet t_1^k$  is again divisible by  $k$ . Continuing in this way, therefore, we find some sequence of (not necessarily mutually distinct) transitions  $\gamma = t_1 \dots t_n \in T^*$  such that  $\Psi(t_1^k \dots t_n^k) = \Psi((\alpha\beta)^k)$  and

$$k \cdot M_0[t_1^k]k \cdot M_1[t_2^k]k \cdot M_2 \dots k \cdot M_{n-1}[t_n^k]k \cdot M_n \text{ with } M_n = M_0.$$

Moreover,  $\Psi(\alpha) \leq \Psi(\gamma)$  because  $\Psi(\alpha^k) \leq \Psi((\alpha\beta)^k) = \Psi(t_1^k \dots t_n^k) = \Psi(\gamma^k)$ . By construction, also,  $M_0[t_1]M_1[t_2]M_2 \dots M_{n-1}[t_n]M_0$ . As  $N$  is persistent by Lemma 1, Keller's theorem can be applied at  $M_0$  in  $N$ . Since  $M_0[\alpha]M$  and  $M_0[\gamma]M_0$ , one obtains both  $M_0[\alpha]M[\gamma \bullet \alpha]M'$  and  $M_0[\gamma]M_0[\alpha \bullet \gamma]M'$ , for some marking  $M'$ . Since  $\Psi(\alpha) \leq \Psi(\gamma)$ , we have  $\alpha \bullet \gamma = \varepsilon$ , and hence  $M' = M_0$ . Thus we have found a sequence  $\beta'$ , namely  $\beta' = \gamma \bullet \alpha$ , leading back from  $M$  to  $M_0$ :  $M_0[\alpha]M[\beta']M_0$ . Since  $\alpha$  was arbitrary,  $N$  is reversible.  $\square$

### 4 The Minimal Cycles of a Reversible and Persistent $k$ -Net

Theorem 2 and Lemmas 1 and 2 imply that a pbrp  $k$ -net with  $n \geq 2$  minimal realizable T-invariants can always be decomposed into  $n$  disjoint pbrp  $k$ -nets, each of which has exactly one minimal realizable T-invariant  $X$ . The latter case is scrutinized in this section, where we will establish the following theorem. Recall that a transition  $t$  is *weakly live* at marking  $M$  if  $\exists M' \in [M]$  such that  $M'(t)$ .

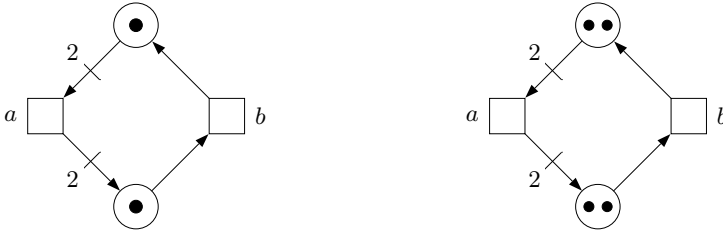
**Theorem 3.** SIMPLE CYCLES IN  $k \cdot N$  HAVE PARIKH VECTOR 1

Let  $k \geq 2$  and let  $(N, k \cdot M_0)$  be a pbrp  $k$ -net with exactly one minimal realizable T-invariant  $X$ . Then  $X \leq 1$  and for any transition  $t$ ,  $X(t) = 0$  if and only if  $t$  is not weakly live at  $k \cdot M_0$ .

In the rest of the section, we assume w.l.o.g. that all transitions are weakly live, and we show that  $X = 1$  under this stronger assumption.

Plainness is important for Theorem 3 to hold. In Figure 3, all simple cycles of the net on the right-hand side have Parikh vector  $X = \Psi(abb)$ , but  $X \neq 1$ , contrary to the conclusion of Theorem 3.

Recall that in a (plain) connected marked graph  $N$ , all transitions occur an equal number of times in any cycle [5,6]. Any such marked graph has thus exactly one minimal realizable T-invariant, viz. the vector 1. Theorem 3 extends this behavioural property of connected marked graphs to pbrp-nets  $k \cdot N$  ( $k \geq 2$ ) with exactly one minimal realizable T-invariant. It is worth noting that the statement made in Theorem 3 would not hold under the weaker assumption that  $N$  instead of  $k \cdot N$  is persistent. For instance, let  $k = 2$  and consider Figure 2. On the left-hand side,  $X = (a \mapsto 1, b \mapsto 1, c \mapsto 2)$  is the unique minimal realizable T-invariant, and it can be realized by the firing sequence  $M_0[acbc]M_0$ .



**Fig. 3.** A weighted Petri net (l.h.s.) and its 2-multiple (r.h.s.)

Note that  $X \neq 1$ . On the right-hand side,  $X$  is also the unique minimal realizable T-invariant. However, the net shown on the right-hand side of Figure 2 is not persistent. Executing  $a$  in the initial marking leads to a marking in which both  $a$  and  $b$  are enabled although their shared input place  $s$  carries only one token, hence producing a true conflict and destroying persistency. Thus, both requirements that  $k \cdot N$  be persistent and that  $k \geq 2$  are crucial for Theorem 3 to hold.

We shall now give the proof of Theorem 3, which is critical to all results established in the remaining sections. By way of approaching this proof, let  $k \cdot N$  be a pbrp-net with exactly one minimal realizable T-invariant  $X$ . By assumption, all transitions are weakly live, hence we want to show  $X = 1$ . As every weakly live transition must occur at least once in any firing sequence realizing  $X$ ,  $X \geq 1$  in view of Theorem 1 and the unicity of  $X$ . If  $X$  is a  $k$ -multiple, then, a contradiction of the assumption  $k \geq 2$  can be derived easily as we will see later. The complicated case is when  $X$  is *not* a  $k$ -multiple. In this case, we will *i*) construct a T-invariant which extends  $X$  and is a  $k$ -multiple; *ii*) show that this new T-invariant is realizable in  $k \cdot N$ ; and *iii*) show that a contradiction to the minimality of  $X$  ensues unless  $X = 1$ . In order to construct this T-invariant, we use the fact that  $X$  is realized by some firing sequence  $\alpha$  in  $T^*$  and we introduce an auxiliary function  $zip_k : T^* \rightarrow T^*$  which, given any sequence  $\alpha \in T^*$ , constructs from  $\alpha$  another sequence  $zip_k(\alpha)$  with  $\Psi(\alpha) \leq \Psi(zip_k(\alpha))$  such that the latter is a  $k$ -multiple. Intuitively,  $zip_k$  yields a “ceiling” operation on Parikh vectors with respect to divisibility by  $k$ .

**Definition 1.** FUNCTION  $zip_k$

Let  $zip_k : T^* \rightarrow T^*$  be the function inductively defined with the following equations, where  $t \in T$ :

$$\begin{aligned} zip_k(\varepsilon) &= \varepsilon \\ zip_k(t\alpha') &= t^k zip_k(\alpha' \bullet t^{k-1}) \end{aligned} \tag{1}$$

□

It follows directly from this definition that  $\Psi(zip_k(\alpha))$  is a  $k$ -multiple and more precisely, the least  $k$ -multiple larger than or equal to  $\Psi(\alpha)$ . Moreover, if  $\Psi(\alpha) \leq k-1$  then  $zip_k(\alpha) = a_1^k \dots a_l^k$ , where  $a_1 \dots a_l$  are all distinct letters of  $\alpha$ , in the order of their first occurrences. For example,  $zip_5(ab^4a^3) = a^5 zip_5(b^4) = a^5 b^5$ .

**Lemma 3.** ENABLING  $zip_k(\alpha)$

Let  $k \cdot N$  be plain and persistent, and let  $k \cdot M$  be a reachable  $k$ -marking of  $k \cdot N$ . If a sequence  $\alpha$  is enabled at  $k \cdot M$ , then  $zip_k(\alpha)$  is also enabled at  $k \cdot M$ .

**Proof:** We use Keller’s theorem and induction on the length of sequences. If  $\alpha = \varepsilon$ , the claim is obviously true.

Suppose now  $\alpha = t\alpha'$  and  $k \cdot M[t\alpha']$ , with  $t \in T$ . By plainness,  $M[t]M'$  and hence  $k \cdot M[t^k]k \cdot M'$ . By Keller’s theorem, also  $k \cdot M[t^k]k \cdot M'[(t\alpha') \bullet (t^k)]$ , and therefore  $k \cdot M[t^k]k \cdot M'[\alpha' \bullet t^{k-1}]$ . By induction hypothesis,  $k \cdot M'[zip_k(\alpha' \bullet t^{k-1})]$ ; hence  $k \cdot M[t^k]k \cdot M'[zip_k(\alpha' \bullet t^{k-1})]$ . By the definition of  $zip_k(\alpha)$ ,  $k \cdot M[zip_k(\alpha)]$ . □

In the sequel, we apply the  $zip_k$  construction to cycles  $k \cdot M_0[\gamma]k \cdot M_0$  of  $k \cdot N$ , and we use the property that if  $\hat{\gamma} = zip_k(\gamma)$ , then the Parikh vector of  $\hat{\gamma}$  may be computed from the Parikh vector of  $\gamma$ . We describe now this computation.

First, we note that the Parikh vector of  $\gamma$  splits (uniquely) as a sum:

$$\Psi(\gamma) = Y_k + Y_{k-1} + \dots + Y_1$$

where  $k|Y_k$  (read  $k$  divides  $Y_k$ ) and for all  $k-1 \geq h \geq 1$  and for all transitions  $t$ ,  $Y_h(t) \in \{h, 0\}$ . Indeed, let  $d_t = \Psi(\gamma)(t) \mathbf{div} k$  (where  $\mathbf{div}$  denotes integer division) and  $h_t = \Psi(\gamma)(t) \mathbf{mod} k$  for every transition  $t$ . Define  $Y_k(t) = k \cdot d_t$  (thus  $k|Y_k(t)$ ), and for  $k-1 \geq h \geq 1$ , define  $Y_{h_t}(t) = h_t$  and  $Y_h(t) = 0$  if  $h \neq h_t$ . We claim that

$$\begin{aligned} \Psi(\hat{\gamma}) &= Y_k + \sum_{h=1}^{k-1} \left(\frac{k}{h}\right) \cdot Y_h \\ &= Y_k + \frac{k}{k-1} \cdot Y_{k-1} + \frac{k}{k-2} \cdot Y_{k-2} + \dots + \frac{k}{2} \cdot Y_2 + k \cdot Y_1 \end{aligned} \tag{2}$$

This can be seen by examining the  $zip_k$  construction. In fact,  $zip_k(\gamma)$  is computed by first moving to the left  $d_t$  subwords  $t^k$  of  $\gamma$  for each transition  $t$  (this does not affect the length of the sequence), and then moving to the left, for each transition  $t$  still appearing on the right, all  $h_t$  occurrences still untouched, augmented with  $k-h_t$  new occurrences of  $t$  if  $h_t \neq 0$  (this may increase the length of the sequence).

**Example** (with  $k = 5$ ):

$$\begin{aligned} zip_5(a^4ba^3) &= a^5 zip_5(ba^2) \quad (\text{the first five } a \text{ s are moved left}) \\ &= a^5 b^5 zip_5(a^2) \quad (\text{one } b \text{ is moved left; four } b \text{ s are added}) \\ &= a^5 b^5 a^5. \quad (\text{two more } a \text{ s are moved left; three } a \text{ s are added}). \end{aligned}$$

Writing Parikh vectors as  $\begin{pmatrix} x \\ y \end{pmatrix}$  to denote entries  $x$  for  $a$  and  $y$  for  $b$ , we have:

$$\begin{aligned} \Psi(a^4ba^3) &= \begin{pmatrix} 7 \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 5 \\ 0 \end{pmatrix}}_{Y_5} + \underbrace{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}_{Y_4} + \underbrace{\begin{pmatrix} 0 \\ 0 \end{pmatrix}}_{Y_3} + \underbrace{\begin{pmatrix} 2 \\ 0 \end{pmatrix}}_{Y_2} + \underbrace{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}_{Y_1} \\ \Psi(a^5b^5a^5) &= Y_5 + \frac{5}{4} \cdot Y_4 + \frac{5}{3} \cdot Y_3 + \frac{5}{2} \cdot Y_2 + 5 \cdot Y_1. \end{aligned}$$

We are now in a position to produce a proof of Theorem 3.

**Proof:** Let  $k \cdot M_0[\gamma]k \cdot M_0$  be a simple cycle in  $k \cdot N$ , thus  $\gamma$  realizes  $X$ . We distinguish two exhaustive and mutually exclusive cases.

**Case 1:**  $k|\Psi(\gamma)$ , that is, all entries of the Parikh vector of  $\gamma$  are divisible by  $k$ .

If  $\gamma = \varepsilon$ , then the net has no transitions and there is nothing to prove.

Otherwise, let  $t$  be the first transition in  $\gamma$ . Because  $\Psi(\gamma)$  is a  $k$ -multiple,  $t$  occurs at least  $k$  times in  $\gamma$ , that is,  $\Psi(t^k) \leq \Psi(\gamma)$ . As  $k \cdot M_0[t]$  and  $k \cdot N$  is a plain net, necessarily  $k \cdot M_0[t^k]k \cdot M_1$  for some  $k$ -multiple marking  $k \cdot M_1$ . By Keller's theorem,  $k \cdot M_1[\gamma \bullet t^k]k \cdot M_0$ . Moreover,  $\Psi(\gamma \bullet t^k)$  is another  $k$ -multiple since  $\Psi(t^k)$  is smaller than or equal to  $\Psi(\gamma)$  and thus,  $\Psi(\gamma \bullet t^k) = \Psi(\gamma) - \Psi(t^k)$ .

Let  $t_1 = t$ . Continuing in this way, we find  $t_2, \dots, t_n$  such that

$$k \cdot M_0[t_1^k]k \cdot M_1[t_2^k] \dots [t_n^k]k \cdot M_0.$$

As  $k \cdot M_0[t_1^k \dots t_n^k]k \cdot M_0$ , by plainness,  $M_0[t_1 \dots t_n]M_0$ , and therefore, *a fortiori*,  $k \cdot M_0[t_1 \dots t_n]k \cdot M_0$ .

Seeing that  $\Psi(t_1 \dots t_n)$  is a realizable T-invariant in  $k \cdot N$ , this Parikh vector must be greater than or equal to  $X$ . Therefore,

$$\Psi(\gamma) = X \leq \Psi(t_1 \dots t_n) = \frac{1}{k}\Psi(t_1^k \dots t_n^k) = \frac{1}{k}\Psi(\gamma),$$

yielding a contradiction since  $k \geq 2$  (and  $\gamma$  is not empty).

**Case 2:**  $k \nmid \Psi(\gamma)$ .

Define  $\widehat{\gamma} = zip_k(\gamma)$ . Let  $\Psi(\gamma) = Y_k + \sum_{h=1}^{k-1} Y_h$  and  $\Psi(\widehat{\gamma}) = Y_k + \sum_{h=1}^{k-1} (\frac{k}{h} \cdot Y_h)$  be the respective decompositions of these two vectors defined above, thus  $Y_k$  is a  $k$ -multiple and for every  $k - 1 \geq h \geq 1$  and  $t \in T$ ,  $Y_{h_t}(t) = h_t = \Psi(\gamma)(t) \bmod k$  and  $Y_h(t) = 0$  for  $h \neq h_t$ . Note that  $Y_{k-1} + \dots + Y_1$  is not the null vector, since  $k \nmid \Psi(\gamma)$ .

From  $k \cdot M_0[\gamma]$  and by Lemma 3,  $k \cdot M_0[\widehat{\gamma}]L$  for some marking  $L$ . As  $k \cdot M_0$  is a  $k$ -marking and  $\Psi(\widehat{\gamma})$  is a  $k$ -multiple,  $L$  is also a  $k$ -marking, say  $L = k \cdot M_1$ . Thus  $k \cdot M_0[\widehat{\gamma}]k \cdot M_1$ . Let  $\gamma$  and  $\widehat{\gamma}$  be renamed  $\gamma_1$  and  $\widehat{\gamma}_1$ , respectively. So far,

$$k \cdot M_0[\widehat{\gamma}_1]k \cdot M_1.$$

By Theorem 1 and the assumption that  $X$  is the only minimal realizable T-invariant of  $k \cdot N$ ,  $k \cdot M_1[\gamma_2]k \cdot M_1$  for some simple cycle with Parikh vector  $\Psi(\gamma_2) = X$ . One may now iterate the construction of  $\widehat{\gamma}_{i+1}$  and  $k \cdot M_{i+1}$  from  $\gamma_{i+1}$  and  $k \cdot M_i$  (presented above for  $i = 0$ ). By doing so, one obtains an infinite sequence

$$k \cdot M_0[\widehat{\gamma}_1]k \cdot M_1[\widehat{\gamma}_2]k \cdot M_2[\widehat{\gamma}_3]k \cdot M_3 \dots$$

where all  $\widehat{\gamma}_i$  have the same Parikh vector as  $\widehat{\gamma}$ , namely the one given by (2), since  $\Psi(\gamma_i) = \Psi(\gamma)$  for all  $i$ . As the net  $k \cdot N$  is bounded, the markings  $k \cdot M_0, k \cdot M_1, \dots$  cannot be all different, hence there exists some finite nonempty subsequence of the form

$$k \cdot M_{i-1}[\widehat{\gamma}_i \widehat{\gamma}_{i+1} \dots \widehat{\gamma}_j]k \cdot M_j, \text{ with } 1 \leq i \leq j \text{ and } k \cdot M_{i-1} = k \cdot M_j.$$

Between  $k \cdot M_{i-1}$  and  $k \cdot M_j$ , there are  $(j - i + 1) \geq 1$  sequences with Parikh vectors equal to  $\Psi(\widehat{\gamma})$ . Thus,  $(j - i + 1) \cdot \Psi(\widehat{\gamma})$  is a realizable T-invariant and necessarily,  $\Psi(\widehat{\gamma})$  also is, showing that  $k \cdot M_0[\widehat{\gamma}]k \cdot M_0$ .

So far, we have constructed two T-invariants,  $\Psi(\gamma)$  and  $\Psi(\widehat{\gamma})$ , such that the latter is a  $k$ -multiple and extends the former, which is not a  $k$ -multiple. The remaining part of the proof contains an elaborate argument showing that this is possible only when  $\Psi(\gamma) = 1$ .

Recall that  $k \cdot M_0[\gamma]k \cdot M_0$ , and  $\Psi(\gamma) \leq \Psi(\widehat{\gamma})$ . By Keller’s theorem,  $k \cdot M_0[\widehat{\gamma} \bullet \gamma]$ , and by  $\Psi(\gamma) \leq \Psi(\widehat{\gamma})$ ,  $\Psi(\widehat{\gamma} \bullet \gamma) = \Psi(\widehat{\gamma}) - \Psi(\gamma)$ . The latter difference is not null, since  $k|\Psi(\widehat{\gamma})$  but  $k \nmid \Psi(\gamma)$  (assumption of Case 2). As  $\Psi(\widehat{\gamma})$  and  $\Psi(\gamma)$  are T-invariants, so is  $\Psi(\widehat{\gamma}) - \Psi(\gamma)$ . Moreover,  $\Psi(\widehat{\gamma} \bullet \gamma) = \Psi(\widehat{\gamma}) - \Psi(\gamma)$  is realizable since  $k \cdot M_0[\widehat{\gamma} \bullet \gamma]$ .

Using equation (2) and  $X = \Psi(\gamma) = Y_k + \dots + Y_1$ , one obtains

$$\Psi(\widehat{\gamma} \bullet \gamma) = \Psi(\widehat{\gamma}) - \Psi(\gamma) = \sum_{h=1}^{k-1} \left( \frac{k-h}{h} \cdot Y_h \right). \tag{3}$$

As  $\Psi(\widehat{\gamma} \bullet \gamma)$  is a realizable T-invariant and  $X (= \Psi(\gamma))$  is the unique minimal realizable T-invariant of  $k \cdot N$ ,  $\Psi(\widehat{\gamma} \bullet \gamma) = l \cdot X$  for some positive integer  $l$ . Thus  $\Psi(\widehat{\gamma}) = \Psi(\widehat{\gamma} \bullet \gamma) + X = (l+1) \cdot X$ . Combining the above, one obtains:

$$\sum_{h=1}^{k-1} \left( \frac{k-h}{h} \cdot Y_h \right) + X = \Psi(\widehat{\gamma}) = l \cdot X + X = l \cdot Y_k + \left( l \cdot \sum_{h=1}^{k-1} Y_h \right) + X.$$

The first equation follows from (3) and from  $\Psi(\gamma) = X$ ; the second equation follows from  $\Psi(\widehat{\gamma}) = (l+1) \cdot X$ ; the third equation follows from  $X = Y_k + \dots + Y_1$ . Comparing the rightmost and leftmost sums in this equation, one gets:

$$l \cdot Y_k = \sum_{h=1}^{k-1} \frac{k - (l+1) \cdot h}{h} \cdot Y_h \tag{4}$$

We show now that  $Y_k$  must be the null vector. For contradiction, assume the contrary. Then  $Y_k(t) \geq 1$  for some transition  $t$ . As  $Y_k$  is a  $k$ -multiple, even  $Y_k(t) \geq k$  and  $l \cdot Y_k(t) \geq l \cdot k$ . As  $l > 0$  and in view of equation (4),  $Y_{h_t}(t) \neq 0$  since by definition of  $Y$ ,  $Y_h(t) = 0$  for any  $1 \leq h \leq k - 1$  with  $h \neq h_t$ . Thus,  $Y_{h_t}(t) = h_t$ . Combining these two properties and remembering that  $k \geq 2$ ,

$$0 < l \cdot k \leq l \cdot Y_k(t) = k - (l+1) \cdot h_t, \tag{5}$$

However,  $1 \leq h_t$  and  $1 \leq l$  entail  $k - (l+1) \cdot h_t \leq k - 2$ , and with (5), one gets  $l \cdot k \leq k - 2$ . As  $l$  is a positive integer, we have reached a contradiction. Thus,  $Y_k$  is indeed the null vector.

$Y_k$  being the null vector means that  $\Psi(\gamma) \leq k - 1$ . Recall that  $\widehat{\gamma} = zip_k(\gamma)$ . By the definition of  $zip_k$  (and the remark just after Definition 1),  $\widehat{\gamma} = t_1^k \dots t_n^k$  where  $t_1 \dots t_n$  are all distinct transitions occurring in  $\gamma$ , with  $t_i \neq t_j$  for  $i \neq j$ . As  $k \cdot M_0[\widehat{\gamma}]k \cdot M_0$ , by plainness  $M_0[t_1 \dots t_n]M_0$ , and a fortiori  $k \cdot M_0[t_1 \dots t_n]k \cdot M_0$ .

Seeing that  $\Psi(t_1 \dots t_n)$  is a realizable T-invariant, this Parikh vector must be greater than or equal to  $X$ . As the transitions  $t_1, \dots, t_n$  are mutually distinct, necessarily  $\Psi(t_1 \dots t_n) \leq 1$ . Therefore,  $1 \leq X \leq \Psi(t_1 \dots t_n) \leq 1$ . Altogether,  $X = 1$  (and also  $\Psi(t_1 \dots t_n) = 1$ ), as was to be shown.  $\square$

As already mentioned, the property stated for pbrp nets  $k \cdot N$  with  $k \geq 2$  in Theorem 3 is a classical property of plain connected marked graphs. A natural question is whether any pbrp net  $k \cdot N$  with exactly one minimal realizable invariant  $X$  can be transformed to a marked graph by just erasing redundant places. The answer to this question is negative; an example is provided in 4.

### 5 Definition of Separability

We distinguish two notions of separability.

**Definition 2.** WEAK AND STRONG SEPARABILITY

Let  $k \geq 1$  and let  $(N, k \cdot M)$  be any net with  $k$ -marking  $k \cdot M$ .

A firing sequence  $k \cdot M[\sigma]$  is *weakly  $k$ -separable* from  $k \cdot M$  (or just weakly separable if  $k$  and  $M$  are understood from the context) if there exist  $k$  sequences  $\sigma_1, \dots, \sigma_k$  such that

$$(\forall j, 1 \leq j \leq k: M[\sigma_j] \text{ in } (N, M)) \quad \text{and} \quad \left(\sum_{j=1}^k \Psi(\sigma_j)\right) = \Psi(\sigma). \tag{6}$$

A firing sequence  $k \cdot M[\sigma]$  is *strongly  $k$ -separable* from  $k \cdot M$  if there exist  $k$  sequences  $\sigma_1, \dots, \sigma_k$  such that

$$(\forall j, 1 \leq j \leq k: M[\sigma_j] \text{ in } (N, M)) \quad \text{and} \quad \sigma \in \sqcup_{j=1}^k \sigma_j, \tag{7}$$

where  $\sqcup$  denotes the shuffle product (“arbitrary interleaving”) operator. A  $k$ -net is weakly (strongly) separable if every sequence firable in its initial marking is weakly (strongly) separable from this  $k$ -marking.  $\square$

**Example:** The 2-net shown in Figure 4 is not strongly 2-separable from the indicated marking  $2 \cdot M$  since  $2 \cdot M[acbbc]$  cannot be obtained by shuffling two firing sequences from  $M$ . However, this 2-net is weakly 2-separable from  $2 \cdot M$ . In particular,  $\Psi(acbbc) = \Psi(abc) + \Psi(abc)$ , and clearly,  $M[abc]$ . The considered 2-net is neither reversible nor persistent; e.g.,  $2 \cdot M[acab]$  and  $2 \cdot M[acac]$  but  $acacb$  cannot be fired from  $2 \cdot M$ .

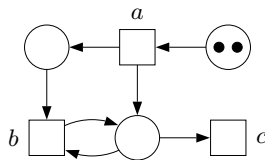


Fig. 4. A weakly but not strongly separable net



## 6 Weak Separability

In this section and in section 7, we will establish the weak (strong, respectively) separability of pbrp-nets under the special assumption that there exists exactly one minimal realizable T-invariant  $X$ . In the rest of this section and in section 7, this assumption applies *implicitly* to all  $k$ -nets under consideration. The results will be extended to the general case in section 8.

As strong separability entails weak separability, one should explain why we examine first weak separability. When weak separability is considered, the freedom to apply permutations to transitions whenever needed allows strong constraints to be imposed on the decompositions of transition sequences into parallel processes; this will later, in Table 1, be made more precise. Such strong constraints will serve to determine at each stage in the inductive decomposition of a transition sequence which process should be extended by the last transition taken into account. For strong separability, one cannot apply permutations to re-arrange processes, and the mathematical structure under the decompositions gets partly obscured. In order to show that inductive decompositions can be obtained with respect to strong separation, and in particular that at each stage in the induction, at least one process can be extended by the last transition taken into account, we shall therefore rely crucially on weak separation.

In the sequel, we usually denote by  $N = (N, M_0)$  the net with initial marking  $M_0$  under consideration, by  $k \cdot N$  the net  $(N, k \cdot M_0)$  with initial  $k$ -marking  $k \cdot M_0$ , and by  $X$  the unique minimal realizable T-invariant of  $k \cdot N$ . Note that if  $k \cdot N$  is a pbrp-net and  $k \geq 2$ , then  $X \leq 1$  by Theorem 3.

### Lemma 4. SHIFTING $k$ -MULTIPLE SUBWORDS

Let  $N$  be plain. Let  $k \geq 2$  and let  $k \cdot N$ , with initial marking  $k \cdot M_0$ , be bounded, reversible, and persistent. Suppose  $k \cdot M_0[\sigma]$ . Then there is some sequence of transitions  $t_1 \dots t_n$  such that

$$k \cdot M_0[t_1^k \dots t_n^k] k \cdot M_1[\sigma'] \text{ with } \Psi(\sigma') \leq k - 1 \text{ and } \sigma \equiv_{k \cdot M_0} t_1^k \dots t_n^k \sigma'.$$

**Proof:** Choose a transition  $t_1$  which is enabled at  $M_0$  and satisfies  $\Psi(\sigma)(t_1) \geq k$ , i.e., such that there are at least  $k$  occurrences of  $t_1$  in  $\sigma$ , if such a transition exists. By plainness and by Keller's theorem,

$$k \cdot M_0[t_1^k] k \cdot M'_0[\sigma \bullet t_1^k].$$

Choose a transition  $t_2$  which is enabled at  $M'_0$  and satisfies  $\Psi(\sigma \bullet t_1^k)(t_2) \geq k$ , if such a transition exists. Again by plainness and by Keller's theorem,

$$k \cdot M_0[t_1^k] k \cdot M'_0[t_2^k] k \cdot M''_0[\sigma \bullet (t_1^k t_2^k)].$$

Repeating this reordering procedure as long as possible, one constructs a sequence

$$k \cdot M_0[t_1^k \dots t_n^k] k \cdot M_1[\sigma']$$

where  $\sigma' = \sigma \bullet (t_1^k \dots t_n^k)$  (possibly  $n = 0$ , in which case  $\sigma' = \sigma$  and  $M_1 = M_0$ ) and  $\Psi(\sigma')(t) \leq k-1$  for every transition  $t$  enabled at  $M_1$ .

We show that no transition (not just the ones enabled at  $M_1$ ) can occur more than  $k - 1$  times in  $\sigma'$ . To this end, let  $k \cdot M_1[\gamma]k \cdot M_1$  be any cycle such that  $\Psi(\gamma) \leq 1$ . Such a cycle must exist because, on the one hand,  $X$  is a realizable T-invariant of  $k \cdot N$  and  $X \leq 1$  by Theorem 3 and on the other hand, this T-invariant can be realized at every reachable marking of  $k \cdot N$  (by Theorem 1). Repeating this cycle  $k - 1$  times gives a cycle  $k \cdot M_1[\gamma^{k-1}]k \cdot M_1$ .

Applying now Keller's theorem to  $k \cdot M_1[\gamma^{k-1}]$  and  $k \cdot M_1[\sigma']$  yields

$$k \cdot M_1[\sigma' \bullet \gamma^{k-1}] \tag{8}$$

If  $\sigma' \bullet \gamma^{k-1} \neq \varepsilon$  then the first transition of  $\sigma' \bullet \gamma^{k-1}$  is firable at  $k \cdot M_1$  (due to 3) and it occurs at least  $k$  times in  $\sigma'$  (due to  $\sigma' \bullet \gamma^{k-1} \neq \varepsilon$  and the fact, stated in Theorem 3 that  $\Psi(\gamma)(t) = 1$  for any transition  $t$  firable at  $k \cdot M_1$ ). This contradicts the fact that the reordering procedure (extracting such  $t^k$  from  $\sigma$ ) has been repeated as long as possible.

Hence  $\sigma' \bullet \gamma^{k-1} = \varepsilon$ , which, by  $\Psi(\gamma) \leq 1$ , implies that  $\sigma'$  contains every transition at most  $k - 1$  times. By construction,  $\sigma \equiv_{k \cdot M_0} t_1^k \dots t_n^k \sigma'$ . This establishes the claims of the lemma.  $\square$

By applying Lemma 4, a sequence  $\sigma$  fired at  $k \cdot M_0$  can be transformed into a permutation-equivalent sequence, viz.  $t_1^k \dots t_n^k \sigma'$ , consisting of an initial segment (leading to  $k \cdot M_1$ ) in which every transition occurs a multiple of  $k$  times (where the  $t_1, \dots, t_n$  are not necessarily all distinct), followed by a tail, denoted by  $\sigma'$ , in which every transition occurs at most  $k - 1$  times. The next lemma, applied with  $j = k - 1$  and  $L = M = M_1$  (thus  $L + j \cdot M = k \cdot M_1[\sigma']$ ), and with  $\tau = \sigma'$  and  $\chi = \varepsilon$  (thus  $k \cdot M_1[\tau]$  and  $k \cdot M_1[\chi]k \cdot M_1$ ), shows that  $\sigma'$  can be further transformed into an initial segment in which every transition occurs *exactly*  $k - 1$  times and a new tail in which every transition occurs *at most*  $k - 2$  times.

**Lemma 5.** SHIFTING  $j$ -MULTIPLE SUBWORDS FOR  $1 \leq j < k$

Let  $N$  be plain. Let  $k \geq 2$  and let  $k \cdot N$ , with initial marking  $k \cdot M_0$ , be bounded, reversible and persistent. Let  $j$  be a fixed number such that  $1 \leq j < k$ . Then the following implication is valid:

**if** a transition sequence  $\tau$  satisfying  $\Psi(\tau) \leq j$  is firable in  $k \cdot N$  at a reachable marking of the form  $L + j \cdot M$ , and if moreover  $(L + j \cdot M)[\chi]k \cdot M$  for some sequence  $\chi$  such that  $\tau$  and  $\chi$  are transition-disjoint,

**then**  $M[t_1 \dots t_p]$  where  $t_1 \dots t_p$  is an enumeration of the set  $\{t_1, \dots, t_p\} = \{t \mid \Psi(\tau)(t) = j\}$ , and  $\tau \equiv_{L + j \cdot M} t_1^j \dots t_p^j \tau'$  for a sequence  $\tau'$  satisfying  $\Psi(\tau') \leq j - 1$  and not containing  $t_1, \dots, t_p$ . Moreover,  $L + j \cdot M[t_1^j \dots t_p^j]L + j \cdot M'[\chi']k \cdot M'$  for some sequence  $\chi'$  such that  $\tau'$  and  $\chi'$  are transition-disjoint.

For explaining the meaning of this lemma, examine the arrows  $\tau$  and  $\chi$  emanating from the North-Western corner, labelled  $L + j \cdot M$ , of Figure 5. According to the lemma, all instances of the transitions  $t_1, \dots, t_p$ , which occur exactly  $j$  times in  $\tau$ , may be shifted towards the beginning, thus forming an initial segment

$t_1^j \dots t_p^j$  after which the residual sequence  $\tau' = \tau \bullet (t_1^j \dots t_p^j)$  is executed. In  $\tau'$ , every transition occurs now at most  $j - 1$  times, and since  $\tau'$  and  $\chi'$  are transition disjoint, the lemma can be applied again to  $\tau'$ ,  $j - 1$  and  $\chi'$ .

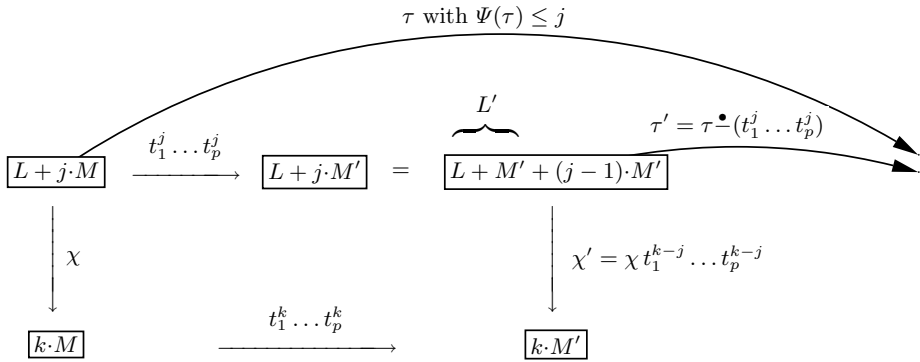


Fig. 5. Explanation of Lemma 5

**Proof:** We use an induction on  $p$ . If  $p = 0$ , then  $\Psi(\tau) \leq j - 1$ , and apart from setting  $\tau' = \tau$ , there is nothing to prove. Otherwise, if  $p > 0$ , we claim that some transition  $t'$  occurring  $j$  times in  $\tau$  is enabled at  $M$  in  $N$ . We establish this claim by producing such  $t'$ .

As  $(L + j \cdot M)[\tau]$  and  $(L + j \cdot M)[\chi]k \cdot M$  in  $k \cdot N$ ,  $(L + j \cdot M)[\chi]k \cdot M[\tau \bullet \chi]$  by Keller's theorem. Therefore, seeing that  $\tau$  and  $\chi$  are transition-disjoint,  $k \cdot M[\tau]$ .

As  $k \cdot M$  is a reachable marking of  $k \cdot N$  and  $X \leq 1$  is the least realizable T-invariant of  $k \cdot N$ , by Theorem 4,  $k \cdot M[\gamma]k \cdot M$  for some sequence  $\gamma$  satisfying  $\Psi(\gamma) = X \leq 1$ . Repeating this cycle  $j - 1$  times yields the cycle  $k \cdot M[\gamma^{j-1}]k \cdot M$ .

By Keller's theorem (applied in  $k \cdot M$  with  $k \cdot M[\tau]$  and  $k \cdot M[\gamma^{j-1}]$ ),  $k \cdot M[\sigma]$  with  $\sigma = \tau \bullet \gamma^{j-1}$ , and since  $\Psi(\tau)(t) \neq 0 \Rightarrow X(t) = \Psi(\gamma)(t) = 1$  (by Theorem 3),  $\Psi(\sigma)(t) = \max\{0, \Psi(\tau)(t) - (j - 1)\}$  for all  $t$ . Now  $\Psi(\tau)(t) = j$  for some  $t$  (since  $p > 0$ ), hence  $\sigma$  differs from the empty sequence. Let  $\sigma = t' \sigma'$ . Then  $k \cdot M[t']$ , hence  $M[t']$  by plainness. Moreover,  $\Psi(\tau)(t') \geq 1 + (j - 1)$ , hence  $\Psi(\tau)(t') = j$ , which establishes our claim.

Let  $t_1 (= t')$  be some transition enabled at  $M$  and occurring  $j$  times in  $\tau$ . Let  $M[t_1]M'$  in  $N$ , then  $(L + j \cdot M)[t_1^j](L + j \cdot M')$  in  $k \cdot N$ . As also  $(L + j \cdot M)[\tau]$ , by Keller's theorem,  $(L + j \cdot M')[\tau']$  with  $\tau' = \tau \bullet t_1^j$ . Thus,  $\Psi(\tau')(t) = \Psi(\tau)(t)$  for  $t \neq t_1$  and  $\Psi(\tau')(t_1) = 0$ , and if we let  $\{t \mid \Psi(\tau)(t) = j\} = \{t_1, \dots, t_p\}$ , then  $\{t \mid \Psi(\tau')(t) = j\} = \{t_2, \dots, t_p\}$ .

In order to get a full proof of the lemma by the induction on  $p$ , it suffices to construct  $\chi'$  such that  $(L + j \cdot M')[\chi']k \cdot M'$  and  $\chi'$  and  $\tau'$  are transition disjoint. We show that both conditions are fulfilled if we set  $\chi' = \chi t_1^{k-j}$ . Transition disjointness is clear since  $t_1$  does not occur in  $\tau' = \tau \bullet t_1^j$  and  $\tau$  and  $\chi$  are transition disjoint. Now  $(L + j \cdot M)[\chi]k \cdot M$ ,  $(L + j \cdot M)[t_1^j](L + j \cdot M')$ , and  $t_1$  does

not occur in  $\chi$  since it occurs in  $\tau$ . By Keller’s theorem and the fundamental equation,  $(L + j \cdot M')[\chi] (L + j \cdot M)' + (k \cdot M - (L + j \cdot M)) = (k - j) \cdot M + j \cdot M'$ . As  $M[t_1] M'$ ,  $(k - j) \cdot M + j \cdot M' [t_1^{k-j}] k \cdot M'$ . Thus, the proof is complete.  $\square$

Iterating the application of Lemma 5 after one application of Lemma 4 is the principle of the proof of our first separability result.

**Theorem 4.** WEAK SEPARABILITY

Let  $N$  be plain. Let  $k \geq 2$  and let  $k \cdot N$ , with initial marking  $k \cdot M_0$ , be bounded, reversible, and persistent. If  $k \cdot N$  has only one minimal realizable  $T$ -invariant, then  $(N, k \cdot M_0)$  is weakly separable.

Note that both reversibility and plainness are important for Theorem 4 to hold. Figure 6 shows on the left-hand side a plain, bounded, non-reversible, persistent Petri net with a 2-marking  $2 \cdot M_0$  such that the firing sequence  $2 \cdot M_0[bcac]$  is not weakly 2-separable. The right-hand side of Figure 6 displays a non-plain, bounded, reversible, persistent 2-net with a 2-marking  $2 \cdot M_0$  in which the firing sequence  $2 \cdot M_0[a]$  cannot be separated for obvious reasons.

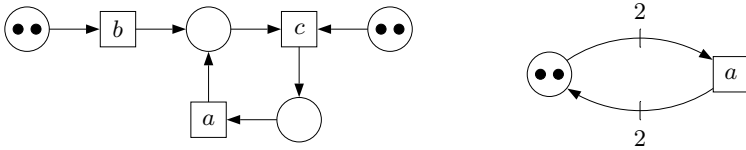


Fig. 6. Two non-separable nets: not reversible (l.h.s.) and not plain (r.h.s.)

**Proof:** Let  $k \cdot M_0[\sigma]$  be given. We show that applying once Lemma 4 and  $k-1$  times Lemma 5 produces a decomposition of  $k \cdot M_0[\sigma]$  into  $k$  sequences  $M_0[\sigma_j]$  ( $j = 1, \dots, k$ ) such that  $\Psi(\sigma) = \sum_{j=1}^k \Psi(\sigma_j)$ . This decomposition is depicted in Table 1, where the  $j$ -th horizontal line shows the “process”  $M_0[\sigma_j]$ . To give a rough idea, the application of Lemma 4 produces the part of the tableau between the first two columns  $M_0 + \dots + M_0$  and  $M_1 + \dots + M_1$ . The  $l$ -th application of Lemma 5 ( $1 \leq l \leq k - 1$ ) produces the part of the tableau between the columns  $M_l + \dots + M_l$  and  $M_{l+1} + \dots + M_{l+1}$ .

We describe now more precisely the successive phases of the decomposition.

**Step 1:** This step consists of applying Lemma 4 to  $k \cdot M_0[\sigma]$ .

The lemma yields  $k \cdot M_0[t_1^k \dots t_n^k] k \cdot M_1[\sigma']$ , with  $\Psi(\sigma') \leq k-1$  and  $\sigma \equiv_{k \cdot M_0} t_1^k \dots t_n^k \sigma'$ . Putting  $n_1 = n$  and  $t_{1,1} = t_1, t_{1,2} = t_2, \dots, t_{1,n_1} = t_n$ , one obtains the part of the tableau to the left of  $M_1 + \dots + M_1$ . (**End of Step 1.**)

**Step 2:** This step consists of  $k-1$  successive applications of Lemma 5 (substeps 2.l for  $l = 1, \dots, k-1$ ).

For every transition  $t$ , let  $h_t = \Psi(\sigma)(t) \bmod k$ , thus  $h_t$  is the remainder left after dividing  $\Psi(\sigma)(t)$  by  $k$ . For each transition  $t$  occurring in  $\sigma'$  (produced in Step 1), if  $h_t = k - l$ , then the  $k - l$  remaining occurrences of  $t$  in  $\sigma'$  are grouped

**Table 1.** A tableau explaining the weak separation of  $\sigma$

$$\begin{array}{ccccccc}
 \sigma_1: & M_0 & \xrightarrow{t_{1,1} \dots t_{1,n_1}} & M_1 & & & \\
 & + & & + & & & \\
 \sigma_2: & M_0 & \xrightarrow{t_{1,1} \dots t_{1,n_1}} & M_1 & \xrightarrow{t_{2,1} \dots t_{2,n_2}} & M_2 & \\
 & + & & + & & + & \\
 \sigma_3: & M_0 & \xrightarrow{t_{1,1} \dots t_{1,n_1}} & M_1 & \xrightarrow{t_{2,1} \dots t_{2,n_2}} & M_2 & \xrightarrow{t_{3,1} \dots t_{3,n_3}} & M_3 \\
 & + & & + & & + & & + \\
 \vdots & \vdots & & \vdots & & \vdots & & \vdots & \dots \\
 & + & & + & & + & & + & \\
 \sigma_k: & M_0 & \xrightarrow{t_{1,1} \dots t_{1,n_1}} & M_1 & \xrightarrow{t_{2,1} \dots t_{2,n_2}} & M_2 & \xrightarrow{t_{3,1} \dots t_{3,n_3}} & M_3 & \dots & \xrightarrow{t_{k,1} \dots t_{k,n_k}} & M_k \\
 & & \underbrace{\hspace{2cm}} & & \underbrace{\hspace{2cm}} & & \underbrace{\hspace{2cm}} & & & \underbrace{\hspace{2cm}} & \\
 & & \tau_1 & & \tau_2 & & \tau_3 & & & \tau_k & \\
 h_t: & & & h_t = k-1 & & h_t = k-2 & & & & h_t = 1 & 
 \end{array}$$

and shifted to the left in the  $l$ -th application (substep 2. $l$ ) of Lemma 5, yielding  $k - l$  subprocesses starting at  $M_l$  and stopping at  $M_{l+1}$ .

More precisely, in substep 2.1, Lemma 5 is applied to

$$\begin{aligned}
 & (L + j \cdot M)[\tau] \quad \text{and} \quad (L + j \cdot M)[\chi]k \cdot M \\
 & \text{with } j = k - 1, \quad L = M_1, \quad M = M_1, \\
 & \quad \tau = \sigma' = \sigma^\bullet (t_{1,1}^k \dots t_{1,n_1}^k), \\
 & \quad \text{and } \chi = \varepsilon.
 \end{aligned}$$

The lemma yields  $M_1[t_1 \dots t_p]$  where  $t_1, \dots, t_p$  is an enumeration of the set  $\{t \mid \Psi(\sigma')(t) = k - 1\}$ , i.e. of the set  $\{t \mid h_t = k - 1\}$ . Putting  $n_2=p$  and  $t_{2,1}=t_1, \dots, t_{2,n_2}=t_p$ , one obtains a decomposition

$$(k - 1) \cdot (M_1[t_{2,1} \dots t_{2,n_2}]M_2) \text{ of } ((k - 1) \cdot M_1)[t_{2,1}^{k-1} \dots t_{2,n_2}^{k-1}]((k - 1) \cdot M_2).$$

In substep 2. $l$  for  $l = 2 \dots, k-1$ , Lemma 5 is similarly applied to

$$\begin{aligned}
 & (L + j \cdot M)[\tau] \quad \text{and} \quad (L + j \cdot M)[\chi]k \cdot M \\
 & \text{with } j = k - l, \quad L = M_1 + \dots + M_l, \quad M = M_l, \\
 & \quad \tau = \sigma^\bullet (t_{1,1}^k \dots t_{1,n_1}^k t_{2,1}^{k-1} \dots t_{2,n_2}^{k-1} \dots t_{l,1}^{k-l+1} \dots t_{l,n_l}^{k-l+1}), \\
 & \quad \text{and } \chi = t_{2,1} \dots t_{2,n_2} \dots t_{3,1}^2 \dots t_{3,n_3}^2 \dots t_{l,1}^{l-1} \dots t_{l,n_l}^{l-1}.
 \end{aligned}$$

**(End of Step 2.)**

Finally, the sequences  $\sigma_1, \dots, \sigma_k$  are defined in accordance with the lines 1 to  $k$  of Table 1. More precisely, for  $1 \leq l \leq k$  let

$$\sigma_l = (t_{1,1} \dots t_{1,n_1}) (t_{2,1} \dots t_{2,n_2}) \dots (t_{l,1} \dots t_{l,n_l}).$$

Then clearly,  $M_0[\sigma_l]M_l$  for  $l = 1, \dots, k$  and  $\Psi(\sigma) = \Psi(\sigma_1) + \dots + \Psi(\sigma_k)$  by construction. Thus, the  $\sigma_1, \dots, \sigma_k$  provide the weak separation of  $\sigma$  that was claimed to exist. □

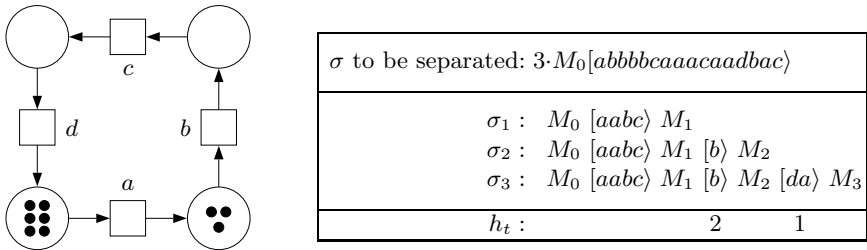


Fig. 7. A 3-net (l.h.s.) and a firing sequence together with a weak separation (r.h.s.)

An example with  $k = 3$  is shown in Figure 7. Consider  $t = a$ . Since  $a$  occurs seven times in  $\sigma$  and  $k = 3$ , we have  $h_a = 1$ . Hence  $a$  occurs (once) in the column determined by  $h_t = 1$ . The remaining six occurrences of  $a$  are spread evenly in the lines between  $M_0 + M_0 + M_0$  and  $M_1 + M_1 + M_1$ . Similarly,  $b$  occurs five times in  $\sigma$ . Thus  $h_b = 2$ , and  $b$  occurs (twice, but only once per line) in the column specified by  $h_t = 2$ .

This example also shows that the weak separation which exists by Theorem 4 is not necessarily a strong separation, since  $\sigma \notin (\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3)$ .

## 7 Strong Separability

Weak separability will now be used in an essential way in order to prove the stronger version, viz. strong separability. In the remainder of this section, we refer to the decomposition constructed in the proof of Theorem 4 and shown in Table 1, relative to a firing sequence  $\sigma$ . In particular,  $M_0, M_1, M_2, \dots, M_k$  refer to the markings shown in this table. To avoid excessive indexing, let  $\tau_i = t_{i,1} \dots t_{i,n_i}$  for  $i = 1, \dots, k$ . Thus  $M_{i-1}[\tau_i]M_i$ , and  $M_0[\sigma_i]M_i$  rewrites as  $M_0[\tau_1]M_1[\tau_2]M_2 \dots M_{i-1}[\tau_i]M_i$ .

Note that any two transitions  $t_{i,j}$  and  $t_{i',j'}$  with  $i, i' \geq 2$  and  $i \neq i'$  or  $j \neq j'$  are different. In particular, also,  $\Psi(\tau_i) \leq 1$  for every  $\tau_i$ .

If some  $k$ -marking enables a transition  $t$ , then in view of the plainness assumption, one  $k$ 'th of this marking also enables  $t$ . We have used this argument several times. The next two lemmata extend this property first from transitions to cycles and next from  $k$ -markings to arbitrary reachable markings.

### Lemma 6. INDIVIDUAL ENABLING PART 1

Let  $N$  be plain. Let  $k \geq 2$  and let  $k \cdot N$  be the multiple of  $N$  with initial marking  $k \cdot M_0$ . Suppose that  $k \cdot N$  is bounded, reversible and persistent, and that  $X \leq 1$  is the unique minimal  $T$ -invariant realized in this net.

If  $k \cdot M_0[\alpha]k \cdot M_0$  is a cycle in  $k \cdot N$  and  $\Psi(\alpha) \leq 1$ , then also  $M_0[\alpha]M_0$  in  $N$ .

**Proof:** Executing  $k$  times the cycle  $\alpha$  in  $k \cdot N$  yields  $k \cdot M_0[\alpha^k]k \cdot M_0$ . Let  $t_1$  be the first transition of  $\alpha^k$  and hence also of  $\alpha$ . Since  $k \cdot M_0[t_1]$ , also  $M_0[t_1]$ , and

then also  $k \cdot M_0[t_1^k]$ . By Keller's theorem,  $k \cdot M_0[t_1^k(\alpha^k \bullet t_1^k)]$ . As  $t_1$  occurs exactly  $k$  times in  $\alpha^k$  (because  $\Psi(\alpha) \leq 1$ ), this firing sequence is of the form:

$$k \cdot M_0[t_1^k] k \cdot M_1[\alpha^k \bullet t_1^k] k \cdot M_0.$$

As  $\Psi(\alpha) \leq 1$ , the first transition of  $\alpha^k \bullet t_1^k$  is also the second transition of  $\alpha$ . Continuing as above, we get a sequence  $t_1 \dots t_n$  of transitions with  $k \cdot M_0[t_1^k \dots t_n^k] k \cdot M_0$ , and then also  $M_0[t_1 \dots t_n] M_0$  in  $N$ , and by construction,  $t_1 \dots t_n = \alpha$ .  $\square$

**Lemma 7.** INDIVIDUAL ENABLING PART 2

Under the same assumptions as in Lemma 6, let  $k \cdot M_0[\sigma]L$  be any firing sequence and let

$$M_0[\sigma_1]M_1, \dots, M_0[\sigma_i]M_i, \dots, M_0[\sigma_k]M_k$$

be the weak separation of this firing sequence given by Table 1 (i.e.,  $L = M_1 + \dots + M_k$  and  $\sigma_i = \tau_1 \dots \tau_i$  with  $\tau_i = t_{i,1} \dots t_{i,n_i}$ ). If  $L[t]$  for some transition  $t$ , then  $M_h[t]$  for some index  $1 \leq h \leq k$ . Moreover, if  $t \neq t_{i,l}$  for all  $i \geq 2$  and  $1 \leq l \leq n_i$  then  $h = k$ , else  $t \in \{t_{h+1,1}, \dots, t_{h+1,n_{h+1}}\}$ .

**Proof:** Suppose that  $L[t]$  with  $t \neq t_{i,j}$  for all  $i \geq 2$  and for all  $j$ . Let  $\tau = \tau_2(\tau_3)^2 \dots (\tau_k)^{k-1}$ , then by construction,  $L[\tau]k \cdot M_k$  (intuitively,  $\tau$  is what is missing in the North-Eastern corner of Table 1). As  $t$  does not occur in  $\tau$ , it follows by persistency that  $k \cdot M_k[t]$ , hence  $M_k[t]$  by plainness.

Suppose that  $L[t]$  with  $t = t_{i,j}$  and  $i \geq 2$ . Then  $t$  occurs in the sequence  $\tau_i$  and in no other  $\tau_{i'}$  with  $i' \neq i$ . As all transitions  $t_{i',j'}$  are different provided that  $i' \geq 2$ ,  $\Psi(\tau_2 \tau_3 \dots \tau_k) \leq 1$ . As  $(N, k \cdot M_0)$  is pbrp,  $(N, k \cdot M_1)$  is pbrp. By Theorem 1, both nets have the same (unique) minimal realizable T-invariant  $X$ , and  $X$  is realized at  $k \cdot M_1$ . By Lemma 6, the T-invariant  $X \leq 1$  (of  $k \cdot N$ ) is realized in  $M_1$  (in  $N$ ). By Theorem 3,  $\Psi(\tau_2 \tau_3 \dots \tau_k) \leq X$ . By Keller's theorem, there must exist a sequence  $\alpha$  such that  $M_k[\alpha]M_1$  and  $\Psi(\tau_2 \tau_3 \dots \tau_k \alpha) = X \leq 1$ . Since  $t$  occurs in  $\tau_i$  and hence also in  $\tau_2 \tau_3 \dots \tau_k$ , it does not occur in  $\alpha$ .

We claim now that

$$L = M_1 + \dots + M_k \begin{matrix} [\tau'] & ((i-1) \cdot M_{i-1} + M_i + \dots + M_k) \\ [\tau''] & ((i-1) \cdot M_{i-1} + (k-i+1) \cdot M_k) \\ [\tau'''] & ((i-1) \cdot M_{i-1} + (k-i+1) \cdot M_1) \\ [\tau'''' ] & k \cdot M_{i-1} \end{matrix}$$

with  $\tau' = \tau_2(\tau_3)^2 \dots (\tau_{i-1})^{i-2}$ ,  $\tau'' = \tau_{i+1}(\tau_{i+2})^2 \dots (\tau_k)^{k-i}$ ,  $\tau''' = \alpha^{k-i+1}$ , and  $\tau'''' = (\tau_2 \dots \tau_{i-1})^{k-i+1}$ .

This may be seen by inspecting Table 1. The sequence  $\tau'$  produces  $i - 1$  copies of  $M_{i-1}$  out of  $M_1 + M_2 + \dots + M_{i-1}$  in the first  $i - 1$  lines of the table. Then  $\tau'' = \tau_{i+1}(\tau_{i+2})^2 \dots (\tau_k)^{k-i}$  produces  $k - i + 1$  copies of  $M_k$  on lines  $i$  to  $k$  of the table. After this,  $k - i + 1$  copies of  $M_1$  are produced by  $\tau''' = \alpha^{k-i+1}$  on lines  $i$  to  $k$ . Finally,  $k - i + 1$  copies of  $M_{i-1}$  are produced by  $\tau''''$  on the same lines.

Now  $L[t]$  and if we let  $\tau = \tau' \tau'' \tau''' \tau''''$ , then  $L[\tau]k \cdot M_{i-1}$  and  $t$  does not occur in  $\tau$  since it appears neither in  $\alpha$  nor in any  $\tau_j$  for  $j \neq i$ . By persistency,  $k \cdot M_{i-1}[t]$ . By plainness,  $M_{i-1}[t]$ .  $\square$

**Theorem 5.** STRONG SEPARABILITY

*Under the same assumptions as in Lemma 6, every firing sequence  $k \cdot M_0[\sigma]L$  has a strong separation.*

**Proof:** We will prove by induction on  $\sigma$  that, if  $k \cdot M_0[\sigma]L$  has the weak separation  $M_0[\sigma_1]M_1, \dots, M_0[\sigma_k]M_k$ , where  $\sigma_i = \tau_1 \dots \tau_i$  and  $\tau_i = t_{i,1} \dots t_{i,n_i}$  as indicated in Table 1 (cf. the proof of Theorem 4), then  $k \cdot M_0[\sigma]L$  belongs to the shuffle of  $k$  firing sequences  $M_0[\zeta_1]M_1, \dots, M_0[\zeta_k]M_k$ , such that  $\Psi(\sigma_i) = \Psi(\zeta_i)$  for all  $i$ .

For  $\sigma$  with length 0, there is nothing to prove. Now let  $\sigma' = \sigma t$  and suppose that the firing sequence  $k \cdot M_0[\sigma]L$  matches both the weak separation  $M_0[\sigma_1]M_1, \dots, M_0[\sigma_k]M_k$  (given by Theorem 4) and the strong separation  $M_0[\zeta_1]M_1, \dots, M_0[\zeta_k]M_k$  (given by induction), such that  $\Psi(\sigma_i) = \Psi(\zeta_i)$  for all  $i$ .

Note that  $\Psi(\tau_1)$  is the integer part of  $\frac{1}{k} \cdot \Psi(\sigma)$  and for  $l > 1$ ,  $\Psi(\tau_l)(t) = 1$  if and only if  $l$  is the rest of the integer division of  $\Psi(\sigma)(t)$  by  $k$ .

The properties under consideration hold clearly for  $\sigma$  with length 0. Assuming they hold for  $\sigma$ , we show now that they hold for  $\sigma' = \sigma t$ . By Theorem 4 and its proof, the firing sequence  $k \cdot M_0[\sigma']$  has a similar weak decomposition  $M_0[\tau'_1]M'_1, \dots, M_0[\tau'_l]M'_l[\tau'_2]M'_2 \dots [\tau'_k]M'_k$ , where  $\Psi(\tau'_1 \dots \tau'_l) = \Psi(\tau_1 \dots \tau_l)$  for all  $l \geq 1$  except one, for which  $\Psi(\tau'_1 \dots \tau'_l) = \Psi(\tau_1 \dots \tau_l) + \Psi(t)$ . Fix this index  $l$ . By the persistency of  $N$  (Lemma 1), and by Keller's theorem, applied to  $M_0[\tau_1 \dots \tau_l]$  and  $M_0[\tau'_1 \dots \tau'_l]$ , necessarily  $M_0[\tau_1 \dots \tau_l t]$ . Therefore,  $M_l[t]$ , showing that one may obtain a strong decomposition of  $k \cdot M_0[\sigma t]$ , i.e. of  $k \cdot M_0[\sigma']$ , by setting  $\zeta'_i = \zeta_i$  for  $i \neq l$  and  $\zeta'_l = \zeta_l t$ . As  $\zeta'_j$  is a permutation of  $\sigma'_j = \tau'_1 \dots \tau'_j$  for all  $j$ , the proof of the theorem follows by the induction on  $\sigma$ . □

The reader may recall from Figure 7 that Theorem 4 does not necessarily yield the sequences  $\zeta_i$  whose shuffle realizes  $\sigma$ . On the other hand, the sequences  $\zeta_i$  yield a weak decomposition of  $\sigma$ , but this weak decomposition does not necessarily enjoy the uniformity and orthogonality properties shown by Table 1.

As an example, consider Figure 8. It shows one step in the proof of Theorem 5, constructing a new strong separation  $\zeta'_j$  and then also a new weak separation  $\sigma'_j$  (of  $\sigma'$ ) from the given separations  $\sigma_j$  and  $\zeta_j$  (of  $\sigma$ ). Note that the initial weak separation is also a strong one, while the new weak separation is no longer strong.

The strong separation  $M_0[\zeta_j]M_j$  of  $k \cdot M_0[\sigma]L$  constructed in the proof of Theorem 5 enjoys the following property, which is fundamental for the simulation of  $k \cdot N$  by  $k$  copies of  $N$ : whenever  $L[t]$  for some transition  $t$ , there exists an index  $l$  such that  $M_l[t]$  and the extension of  $\zeta_l$  by  $t$  gives again a strong separation with this property. It is worth noting that some other possible decompositions of firing sequences with respect to strong separation do not enjoy this property. Consider e.g. the 2-net with the marking  $2 \cdot M$  shown in Figure 9. Then  $2 \cdot M[ab]$  may be decomposed into  $M[a]$  and  $M[b]$ , but  $2M[abc]$  and neither  $M[ac]$  nor  $M[bc]$ .



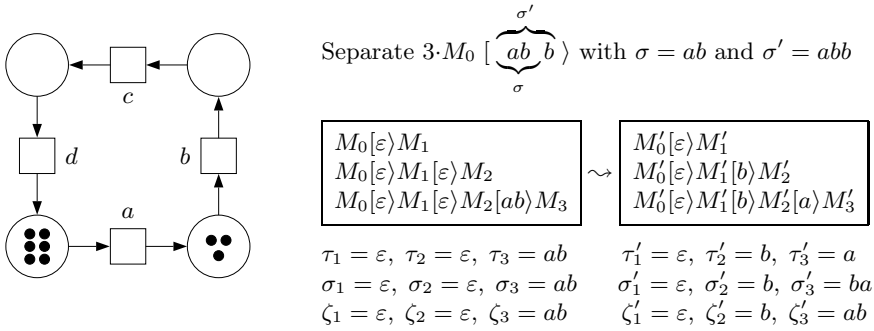


Fig. 8. Illustration of the proof of Theorem 5

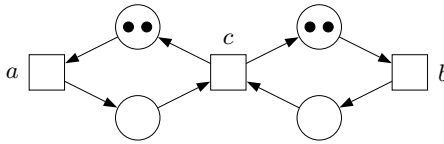


Fig. 9. A marked graph

## 8 The General Case

With the help of Theorem 2, we can now extend the strong separability result to pbrp-nets with several incomparable realizable T-invariants.

### Theorem 6. STRONG SEPARABILITY (FOR GENERAL PBRP-NETS)

Let  $N$  be plain. Let  $k \geq 2$  and let  $k \cdot N$ , with initial marking  $k \cdot M_0$ , be bounded, reversible, and persistent. Then  $(N, k \cdot M_0)$  is strongly separable.

**Proof:** Let  $\{X_1, \dots, X_n\}$  be the set of mutually transition-disjoint T-invariants of  $k \cdot N$  given by Theorem 1. According to Theorem 2, there are  $n$  bounded, reversible and persistent nets  $k \cdot N_1, \dots, k \cdot N_n$  such that the reachability graph of  $k \cdot N$  is isomorphic to the reachability graph of the disjoint sum of the nets  $k \cdot N_1, \dots, k \cdot N_n$ . Moreover, these nets  $k \cdot N_i$  are given by  $k \cdot N_i = (P, T_i, F_i, k \cdot M_0)$  where  $T_i = \{t \in T \mid X_i(t) \neq 0\}$  and  $F_i$  is the induced restriction of  $F$  on  $(P \times T_i) \cup (T_i \times P)$ . Thus all nets  $k \cdot N_i$  have similar initial markings  $k \cdot M_0$  (but for separate copies of the set of places  $P$ ), and  $\{T_1, \dots, T_n\}$  is a partition of the set of transitions  $T$ .

Let  $k \cdot M_0[\sigma]$  be a given firing sequence of  $k \cdot N$ . For  $i = 1, \dots, n$ , let  $\sigma_i$  be the projection of  $\sigma$  on  $T_i^*$ . Thus,  $\sigma \in \bigsqcup_{i=1}^n \sigma_i$ , and in particular,  $\Psi(\sigma) = \sum_{i=1}^n \Psi(\sigma_i)$ . In view of the isomorphism of reachability graphs described above, there must exist corresponding firing sequences  $k \cdot M_0[\sigma_i]$  of nets  $k \cdot N_i$ .

Consider some fixed net  $k \cdot N_i$ . As  $k \cdot N_i$  is the induced (subnet) restriction of  $k \cdot N$  on  $P$  and  $T_i$ , and both nets have the same initial marking, the reachability graph of  $k \cdot N_i$  embeds into the reachability graph of  $k \cdot N$ , and it is isomorphic to the reachable restriction of this labelled graph induced on the subset of labels  $T_i$ . Therefore, the T-invariant  $X_i$  which is realized at  $k \cdot M_0$  in  $k \cdot N$  is also realized at  $k \cdot M_0$  in  $k \cdot N_i$ . Moreover, it is the only minimal realizable T-invariant of  $k \cdot N_i$ . Indeed, any T-invariant which is realized in  $k \cdot N_i$  is also realized in  $k \cdot N$  due to the embedding of reachability graphs, and we know from Theorem 1 that  $X_i$  is the only minimal realizable  $T_i$ -invariant of  $k \cdot N$ . Now,  $k \cdot N_i$  is bounded, reversible and persistent, and it is moreover a  $k$ -net since it has the initial marking  $k \cdot M_0$ . By Theorem 5,  $k \cdot N_i$  is strongly separable, hence there exist  $k$  firing sequences  $M_0[\sigma_{i,1}], \dots, M_0[\sigma_{i,k}]$  of the net  $N_i = (P, T_i, F_i, M_0)$  such that  $\sigma_i \in \bigsqcup_{j=1}^k \sigma_{i,j}$  for each  $i$  from 1 to  $n$ . Thus,  $\sigma \in \bigsqcup_{i=1}^n \bigsqcup_{j=1}^k \sigma_{i,j}$ . By associativity and commutativity of the shuffle product,  $\sigma \in \bigsqcup_{j=1}^k \bigsqcup_{i=1}^n \sigma_{i,j}$ , hence one may choose specific words  $\tau_j \in \bigsqcup_{i=1}^n \sigma_{i,j}$  ( $j = 1, \dots, k$ ) such that  $\sigma \in \bigsqcup_{j=1}^k \tau_j$ . In order to complete the proof of the theorem, it suffices to show that  $M_0[\tau_j]$  in  $N = (P, T, F, M_0)$  for each  $j$  from 1 to  $k$ . Fix  $j$  with  $1 \leq j \leq k$ . As  $k \cdot N$  is bounded, reversible and persistent, by Lemmata 1 and 2,  $N$  enjoys similar properties. For  $i = 1, \dots, n$ , as  $k \cdot N_i$  is bounded, reversible and persistent, by Lemmata 1 and 2,  $N_i$  enjoys similar properties. Therefore, by Theorem 2, the reachability graph of  $N$  (with initial marking  $M_0$ ) is isomorphic to the reachability graph of the disjoint sum of nets  $N_1 + \dots + N_n$  (each of them also with the initial marking  $M_0$ ). In view of this isomorphism, as  $\tau_j$  projects (on  $T_i^*$ ) to  $\sigma_{i,j}$  and  $M_0[\sigma_{i,j}]$  in  $(N_i, M_0)$  for all  $i$  with  $1 \leq i \leq n$ , necessarily,  $M_0[\tau_j]$  in  $N$ .  $\square$

## 9 A Supplementary Result and Some Open Questions

We finally state a further structural result. For its proof, see 4.

**Theorem 7.** *Let  $N$  be plain. Let  $k \geq 2$  and let  $k \cdot N$  be the multiple of  $N$  with initial marking  $k \cdot M_0$ . Suppose that  $k \cdot N$  is reversible, bounded and persistent. Then  $(k - 1) \cdot N$  is weakly separable, pbrp, and strongly separable.*

Several questions remain open. For a  $k$ -net satisfying the preconditions of Theorem 3, does there exist a language-equivalent marked graph? Can Theorem 3 be generalised to initial markings putting either 0 or  $\geq 2$  tokens on each place? Under which conditions can the reversibility assumption be weakened?

Another interesting question is to what extent our results may be used for the simulation of systems. In some sense, they provide both a sequential simulation of  $k$  parallel net systems  $N$  by a net system  $k \cdot N$ , and a parallel simulation of  $k \cdot N$  by  $k$  parallel net systems  $N$ . What may limit the impact is that, at present, we do not know how deciding or checking efficiently that  $k \cdot N$  is pbrp (even knowing that  $N$  is pbrp), which is a precondition for the results to apply.

## Acknowledgements

The first author would like to thank the Université de Rennes 1 for inviting him at IRISA during February 2009. The authors would also like to thank the reviewers for their comments.

## References

1. Best, E., Darondeau, P., Wimmel, H.: Making Petri Nets Safe and Free of Internal Transitions. *Fundamenta Informaticae*, 1–16 (2007)
2. Best, E., Esparza, J., Wimmel, H., Wolf, K.: Separability in Conflict-free Petri Nets. In: Virbitskaite, I., Voronkov, A. (eds.) *PSI 2006*. LNCS, vol. 4378, pp. 1–18. Springer, Heidelberg (2007)
3. Best, E., Darondeau, P.: A Decomposition Theorem for Finite Persistent Transition Systems. *Acta Informatica* 46, 237–254 (2009)
4. Best, E., Darondeau, P.: Separability in Persistent Petri Nets. TR 04/09, Dep. Comp. Sci., Univ. Oldenburg (December 2009),  
<http://parsys.informatik.uni-oldenburg.de/~best/publications/EB-PhD-sep-long.pdf>
5. Commoner, F., Holt, A.W., Even, S., Pnueli, A.: Marked Directed Graphs. *J. Comput. Syst. Sci.* 5(5), 511–523 (1971)
6. Genrich, H.J., Lautenbach, K.: Synchronisationsgraphen. *Acta Informatica* 2(2), 143–161 (1973)
7. van Hee, K., Sidorova, N., Voorhove, M.: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In: van der Aalst, W.M.P., Best, E. (eds.) *ICATPN 2003*. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003)
8. Keller, R.M.: A Fundamental Theorem of Asynchronous Parallel Computation. In: Tse-Yun, F. (ed.) *Parallel Processing*. LNCS, vol. 24, pp. 102–112. Springer, Heidelberg (1975)

# New Algorithms for Deciding the Siphon-Trap Property

Olivia Oanea, Harro Wimmel, and Karsten Wolf

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany  
{olivia.oanea,harro.wimmel,karsten.wolf}@uni-rostock.de

**Abstract.** The siphon-trap property, also known as Commoner-Hack property, establishes a relation between structural entities within a Petri net – the eponymous siphons and traps. The property is linked to the behavior of a Petri net, for instance to deadlock freedom and liveness of the net. It is nevertheless nontrivial to decide the property as a net can have exponentially many siphons and traps even if only minimal siphons are considered. Consequently, the value of the property depends on the availability of powerful decision procedures.

We contribute to this issue by proposing two new methods for deciding the siphon-trap property. One is a plain translation of the property into a Boolean satisfiability (SAT) problem, which exploits the fact that incredibly powerful SAT solvers are available. The second procedure has a divide-and-conquer nature which builds upon a decomposition of a Petri net into *open nets* and projects information about siphons and traps onto the interfaces of the components.

**Keywords:** Petri nets, Traps, Siphons, Commoner-Hack, Liveness, SAT, Divide-and-Conquer.

## 1 Introduction

The siphon-trap property [5,2] is a classical structural property of Petri nets. It states that every siphon (a set of places that cannot switch from unmarked to marked) includes a marked trap (a structure that cannot switch from marked to unmarked). The property can be used for deciding liveness in free choice Petri nets and as a sufficient condition for deadlock freedom in general Petri nets. According to common belief, the main advantage of structural techniques is that they avoid the generation of a state space which is subject to the state explosion problem. In fact, the siphon-trap property involves the investigation of only finitely many finite siphons in the net even for unbounded Petri nets, i.e. infinite state systems. Nevertheless, evaluating the property is far from trivial. Existing tools like INA [6] enumerate potentially exponentially many siphons and may thus run into severe run time and space problems.

We propose two new approaches for evaluating the siphon-trap property of place-transition nets. The first approach translates the property into a Boolean satisfiability problem. Our translation improves results in [10,1] where the property was translated into a Horn-satisfiability problem for bounded free-choice and

other subclasses of Petri nets. The translation as such can be done in polynomial time resulting in a formula with  $n(n + 1)$  propositions, where  $n$  is the number of places. The subsequent satisfiability problem is NP-complete but there is a number of tools available which are capable of solving incredibly large instances in reasonable time.

The second approach follows the divide-and-conquer paradigm. We decompose a Petri net into *open net* components where an open net is a place bordered subnet such that each place on the border (we shall call them *interface places*) represents unidirectional asynchronous communication with exactly one other component. We improve an existing decomposition technique in two directions. First, we present a more efficient algorithm. Second, we propose a net transformation which allows us to divide a net into arbitrarily small components. For each component, information about siphons, traps, and their mutual relation is condensed into constraints for the interface places. Upon composition of components, information of the components is aggregated to corresponding information about the composite open net. Since the size of the condensed information has a stronger correlation to the number of *interface* places than to the overall number of places in a component, the approach has the potential of outperforming traditional algorithms at least for a significant class of nets. This in turn is sufficient for including an algorithm into a tool as present day computing environments support the parallel execution of several tasks.

## 2 Basic Definitions

**Definition 1 (Petri net).** An (unmarked) net is a triple  $(S, T, F)$  where  $S$  and  $T$  are finite sets with  $S \cap T = \emptyset$ , and  $F$  is a mapping  $F: (S \times T) \cup (T \times S) \rightarrow \{0, 1\}$ , i.e. we consider nets without arc weights.

For any unmarked net  $(S, T, F)$  and any  $x \in S \cup T$ , let  $\bullet x := \{y \mid F(y, x) \neq 0\}$  and  $x^\bullet := \{y \mid F(x, y) \neq 0\}$  be the preset and postset of  $x$ , respectively. We extend this notion to sets  $X \subseteq S \cup T$  by  $\bullet X := \bigcup_{x \in X} \bullet x$  and  $X^\bullet := \bigcup_{x \in X} x^\bullet$ . We assume nets have no isolated places, i.e. places  $s$  with  $\bullet s \cup s^\bullet = \emptyset$ .

A marking of  $(S, T, F)$  is a function  $m: S \rightarrow \mathbb{N}$ . We say that a place  $s$  has  $k$  tokens under  $m$  if  $m(s) = k$ . For  $S' \subseteq S$  we introduce the abbreviation  $m(S') := \sum_{s \in S'} m(s)$  and say that  $S'$  is marked under  $m$  iff  $m(S') > 0$ , otherwise it is unmarked.

A marked net is a tuple  $(S, T, F, m_0)$  consisting of an unmarked net  $(S, T, F)$  and an (initial) marking  $m_0$ . An open net  $(S, T, F, m_0, S_i, S_o)$  contains a marked net  $(S, T, F, m_0)$ , a set  $S_i$  of input places with  $S_i \subseteq S$  and  $\bullet S_i = \emptyset$ , a set of output places  $S_o$  with  $S_o \subseteq S$  and  $S_o^\bullet = \emptyset = S_i \cap S_o$ . The set  $I := S_i \cup S_o$  is called the interface of the net, places in  $S \setminus I$  are called inner places. Nets with an empty interface or without an interface at all are called closed nets.

Open nets can be seen as partial nets mergeable via parts of their interfaces using a composition operator  $\oplus$ .

**Definition 2 (Composition of open nets).** For  $k \in \{1, 2\}$  let  $N_k = (S_k, T_k, F_k, m_k, S_{i,k}, S_{o,k})$  be open nets such that  $T_1 \cap T_2 = \emptyset$ ,  $S_{i,1} \cap S_{i,2} = \emptyset = S_{o,1} \cap S_{o,2}$ ,

and  $S_1 \cap S_2 = (S_{i,1} \cap S_{o,2}) \cup (S_{i,2} \cap S_{o,1})$ , i.e. common elements of the two open nets are non-inner places only, and these must be input in one and output in the other open net. Furthermore, for all  $s \in S_1 \cap S_2$ :  $m_1(s) = m_2(s)$  must hold. Then we define  $N_1 \oplus N_2 := (S_1 \cup S_2, T_1 \cup T_2, F_1 \cup F_2, m_1 \cup m_2, S_i, S_o)$ , where  $S_i = (S_{i,1} \cup S_{i,2}) \setminus (S_1 \cap S_2)$  and  $S_o = (S_{o,1} \cup S_{o,2}) \setminus (S_1 \cap S_2)$ .

Note that  $m_1 \cup m_2$  is well-defined since  $m_1$  and  $m_2$  are equal for common places. The composition  $\oplus$  is obviously commutative. Associativity is also easy to see, we notice that a place may appear in the open nets of a well-defined expression of the form  $N_1 \oplus N_2 \oplus N_3 \oplus \dots$  either twice (once as input and once as output place, to be merged to one inner place) or once (as input, output or inner place) or not at all. Matching input and output places in different ways depending on the order of nets is therefore impossible and we can conclude:

**Proposition 1.** *The composition  $\oplus$  is commutative and associative.*

Our main consideration are traps and siphons. A trap is a set of places that cannot be emptied once it contains a token, no matter which transitions fire. A siphon is a set of places that cannot obtain new tokens once it has been emptied of tokens.

**Definition 3 (Traps and siphons).** A trap  $Q$  of an (unmarked or marked) net  $(S, T, F, m_0)$  is a set  $Q \subseteq S$  with  $Q \neq \emptyset$  and  $Q^\bullet \subseteq {}^\bullet Q$ . Analogously, a siphon is a set  $D \subseteq S$  with  $D \neq \emptyset$  and  ${}^\bullet D \subseteq D^\bullet$ . A trap  $Q$  is marked if  $\exists s \in S: m_0(s) > 0$ . For a set  $X$  of places of an open net  $(S, T, F, m_0, S_i, S_o)$ , call  $I(X) = X \cap (S_i \cup S_o)$  the interface of  $X$ . Let such a set be closed if  $I(X) = \emptyset$ , otherwise open. Let a siphon (or trap, resp.)  $M$  be  $X$ -minimal iff  $X \subseteq M$  and no other siphon  $D$  (or trap, resp.) fulfills  $X \subseteq D \subset M$ . For a net  $N$  let  $\mathcal{Q}(N)$  denote the set of all traps in  $N$  and  $\mathcal{D}(N)$  the set of all siphons in  $N$ .

A net  $N = (S, T, F, \dots)$  is called a free-choice net if for each pair  $t, t' \in T$ ,  ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$  implies  ${}^\bullet t = {}^\bullet t'$ . For these free-choice nets there is a well known relation between traps/siphons and liveness, i.e. whether all transitions can be enabled from all reachable markings.

**Proposition 2 (Commoner-Hack [5,2]).** *Let  $N$  be a marked free-choice net. Then  $N$  is live if and only if every siphon of  $N$  contains a marked trap.*

If we consider general nets we can only conclude:

**Proposition 3.** *Let  $N$  be a marked net.*

- (1) *If  $N$  is live then every siphon of  $N$  contains a marked trap.*
- (2) *If every siphon of  $N$  contains a marked trap then  $N$  does not contain deadlocks (i.e. all reachable markings enable at least one transition).*

In the sequel, we shall refer to the property “every siphon contains a marked trap” as the *siphon-trap property* (STP). The remainder of this article is devoted to new decision procedures for the property.

### 3 Evaluating the Siphon-Trap Property Using SAT

In this section we propose a reduction of STP to the famous SAT problem [3]. We aim at a formula that is satisfiable if and only if there is a siphon which does not contain a marked trap. Our starting point is a formula which operates on the places as propositions and whose satisfying assignments correspond exactly to the siphons of a given net. Such formula is well known.

**Lemma 1** ([10,7]). *A set  $D$  of places of a net  $N$  is a siphon if and only if the assignment  $\beta$  with  $\beta(s) = \text{true}$  if and only if  $s \in D$  satisfies*

$$\bigvee_{s \in S} s \wedge \bigwedge_{t \in T} \bigwedge_{s \in t^\bullet} (s \implies \bigvee_{s' \in t^\bullet} s').$$

The first part of the formula states the non-emptiness while the second part is the siphon condition  ${}^\bullet D \subseteq D^\bullet$ . A dual formula is capable of describing traps but can not immediately be used for formulating the STP. The reason is that there is a change of quantifiers: there *exists* a siphon  $D$  such that *every* included trap is unmarked. Hence we use another approach exploiting the fact that every siphon  $D$  containing traps has a unique maximal trap (which is the union of all traps included in  $D$ ). Beginning with a siphon  $D$ , its maximal included trap can be computed by a repeated removal of places  $s$  where some post-transition has no post-place in the so far remaining set. Let  $n$  be the number of places in  $N$ . We represent the repetition of the procedure by introducing  $(n + 1)$  variables  $s^{(0)}, \dots, s^{(n)}$  for each place  $s$ . The variables  $s^{(0)}$  represent a non-empty siphon as mentioned above. The variables  $s^{(i)}$  represent intermediate stages  $D_i$  of the procedure for generating the maximal included trap.  $D_{i+1}$  is obtained from  $D_i$  by removing all places for which some post-transition does not have any post-place in  $D_i$ . Since there are only  $n$  places, the procedure converges after at most  $n$  iterations, so  $D_n$  is either empty or the maximal trap included in  $D$ . The relation between  $D_i$  and  $D_{i+1}$  can be expressed for each place  $s$  individually as follows:

$$s^{(i+1)} \iff (s^{(i)} \wedge \bigwedge_{t \in T} \bigwedge_{s \in t^\bullet} \bigvee_{s' \in t^\bullet} s'^{(i)}).$$

As we want to have the formula satisfied iff the maximal trap is unmarked or non-existent, we add the formula

$$\bigwedge_{s \in S: m_0(s) > 0} \neg s^{(n+1)}.$$

From these considerations, the following theorem is evident.

**Theorem 1.** *In a given net  $N$  with  $n$  places, there exists a siphon which does not include a marked trap if and only if the following formula is satisfiable:*

$$\phi ::= \bigvee_{s \in S} s^{(0)} \wedge \bigwedge_{t \in T} \bigwedge_{s \in t^\bullet} (s^{(0)} \implies \bigvee_{s' \in t^\bullet} s'^{(0)}) \quad (1)$$

$$\wedge \bigwedge_{i=0}^n \bigwedge_{s \in S} (s^{(i+1)} \iff (s^{(i)} \wedge \bigwedge_{t \in T} \bigwedge_{s \in t^\bullet} \bigvee_{s' \in t^\bullet} s'^{(i)})) \quad (2)$$

$$\wedge \bigwedge_{s \in S: m_0(s) > 0} \neg s^{(n+1)} \quad (3)$$

**Table 1.** Evaluating STP: SAT vs. INA

ID	$ P $	$ T $	$ F $	SAT	INA
phils10	50	40	120	0.05 sec	3 sec
phils20	100	80	240	0.24 sec	$\geq 2h$
phils50	250	200	600	2.29 sec	n.a.
phils100	500	400	1200	12 sec	n.a.
phils150	750	600	1800	40 sec	n.a.
phils200	1000	800	2400	119 sec	n.a.
data1010	50	40	300	0.12 sec	8 sec
data1212	60	48	408	0.19 sec	16 sec
data1515	75	60	600	0.36 sec	28 sec

The formula contains  $n(n + 1)$  different propositions, one for each place and iterative step (counted by  $t$ ), and has obviously a length that is polynomial in  $card(S) + card(T) + card(F)$ .

We have implemented an ad-hoc translation from a Petri net to the mentioned formula and shipped it to the state-of-the-art SAT checker MiniSat [9] and compared our results with the STP check done by INA [6]. We obtained the results listed in Table 1. As experimental data, we used the  $k$  dining philosophers examples and the semaphore based scheme for concurrent read and exclusive write access to a database with  $k$  writing and  $k$  reading processes. Observe that the INA check time explodes for the 20 philosophers example while the SAT check has a significant time increase for the 200 philosophers example.

## 4 Evaluating the Siphon-Trap Property Using a Divide-and-Conquer Approach

Deciding liveness is co-NP-complete for free-choice nets according to Esparza and Nielsen [4], so a general fast algorithm is impossible. In the following, we develop an algorithm for evaluating the STP using a divide-and-conquer strategy. The complexity of this algorithm depends more on the size of interfaces during the conquer part than on the size of the nets. Managing to keep the interfaces small may thus lead to a fast algorithm. The general algorithm will look like this:

1. Decompose a (marked) net  $N = (S, T, F, m_0)$  into a set of open net components.
2. Calculate traps and siphons for each such component. For closed siphons, the STP is evaluated using any traditional algorithm.
3. Condense information about open siphons and included traps such that it only refers to the interface.
4. Aggregate components step by step. From the information provided by the components, reason about siphons that become closed through the aggregation and derive information about open siphons and included traps of the aggregated open net.



In Subsection 4.1, we propose a procedure that is able to decompose a Petri net into arbitrarily small open nets. How far to break down a Petri net is optional though. Subsection 4.2 studies the relations between siphons and traps on one hand and open net composition on the other. In Subsection 4.3 we define a structure that is later on used for representing the information about open siphons and traps. Then, we take this information for reasoning about siphons and traps that are closed by aggregation. Finally, we deal with the generation of information about open siphons and traps in an aggregation.

### 4.1 Decomposition into Open Nets

So far we have talked about some aspects of components but we have not defined them yet. Thanks to the composition  $\oplus$ , this is easy to do.

**Definition 4 (Components).** *Let  $N$  be a marked net and  $N_1$  be an open net. We call  $N_1$  a component of  $N$  if there is some open net  $N_2$  with  $N = N_1 \oplus N_2$ .*

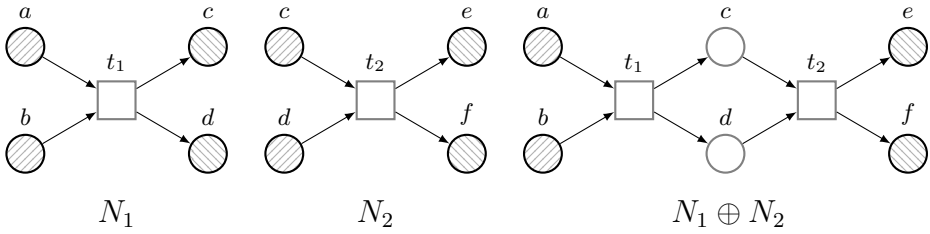
For our divide-and-conquer approach we are usually interested in small components, i.e. we would like to split a net into as many components as possible. Zaitsev [11] presented an algorithm to obtain the unique set of smallest components into which a net can be decomposed. Later, the algorithm was improved by Mennicke et al. [8]. Its idea is to start at some transition and recursively tag necessary net elements until a component is completed:

**Definition 5 (Building components).** *Let  $N = (S, T, F, m_0, S_i, S_o)$  be an open net and  $t \in T$  a transition. The component  $C(t) = (S', T', F|_{(S' \times T' \cup T' \times S')}, m_0|_{S'}, S'_i, S'_o)$  is the smallest (wrt. set inclusion) open net fulfilling the following criteria:*

- (1)  $t \in T'$ ,
- (2) if  $t' \in T'$  then  $\bullet t' \cup t' \bullet \subseteq S'$ ,
- (3) if  $t' \in T'$  then  $(\bullet t') \bullet \cup \bullet (t' \bullet) \subseteq T'$ ,
- (4) for  $s \in S'$ :  $(s \in S_i \vee \exists t' \in T \setminus T' : t' \in \bullet s) \implies s \in S'_i$ ,
- (5) for  $s \in S'$ :  $(s \in S_o \vee \exists t' \in T \setminus T' : t' \in s \bullet) \implies s \in S'_o$ .

Any open net can be disassembled into a set of at most  $|T|$  different components (one for each transition, but  $t' \in C(t)$  implies  $C(t) = C(t')$ ). Different components have disjoint sets of transitions and inner places. Interface places may be shared by components, but each such place may appear only once as input place and once as output place in all components together. Clearly,  $\bigoplus_t C(t) = N$  if we add only one of  $C(t)$ ,  $C(t')$  whenever  $C(t) = C(t')$ .

*Example.* There are nets which can be split up into components with only one transition in each. Take e.g. a cycle of alternating places and transitions, with two places before and after each transition, and one transition before and after each place. All components look alike, the first two being  $N_1$  and  $N_2$  of Fig. 11. Composing further components to the resulting net  $N_1 \oplus N_2$  may prolong the strand until the final component with output places  $a$  and  $b$  is added to complete the cycle.



**Fig. 1.** Two components  $N_1$  and  $N_2$  and their composition  $N_1 \oplus N_2$ . Input places have stripes going upwards, output places downwards.

Since the components are so small, we can easily determine all traps and siphons, e.g. for  $N_1$ :  $\mathcal{Q} = \{\{c\}, \{d\}, \{c, d\}, \{a, c\}, \{b, c\}, \{a, b, c\}, \{a, d\}, \{b, d\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}, \{a, b, c, d\}\}$  and  $\mathcal{D} = \{\{a\}, \{b\}, \{a, b\}, \{a, c\}, \{a, d\}, \{a, c, d\}, \{b, c\}, \{b, d\}, \{b, c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, b, c, d\}\}$ .

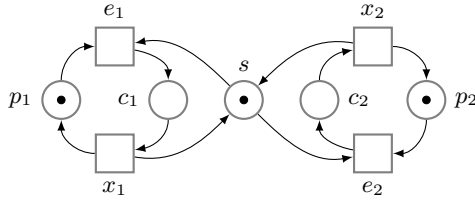
If we restrict ourselves e.g. to  $\{s\}$ -minimal traps and siphons for some place  $s \in S$ , we get the even smaller sets  $\mathcal{Q}_1 = \{\{c\}, \{d\}, \{a, c\}, \{b, c\}, \{a, d\}, \{b, d\}\}$  and  $\mathcal{D}_1 = \{\{a\}, \{b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\}$ .

The conquer part of our divide-and-conquer strategy should later show the siphons and traps of  $N_1 \oplus N_2$  to be  $\mathcal{Q}' = \{\{e\}, \{f\}, \{c, e\}, \{d, e\}, \{c, f\}, \{d, f\}, \{a, c, e\}, \{b, c, e\}, \{a, d, e\}, \{b, d, e\}, \{a, c, f\}, \{b, c, f\}, \{a, d, f\}, \{b, d, f\}\}$  and  $\mathcal{D}' = \{\{a\}, \{b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{a, c, e\}, \{a, c, f\}, \{a, d, e\}, \{a, d, f\}, \{b, c, e\}, \{b, c, f\}, \{b, d, e\}, \{b, d, f\}\}$  (again with the reduction to  $s$ -minimal elements for  $s \in S$ ).  $\square$

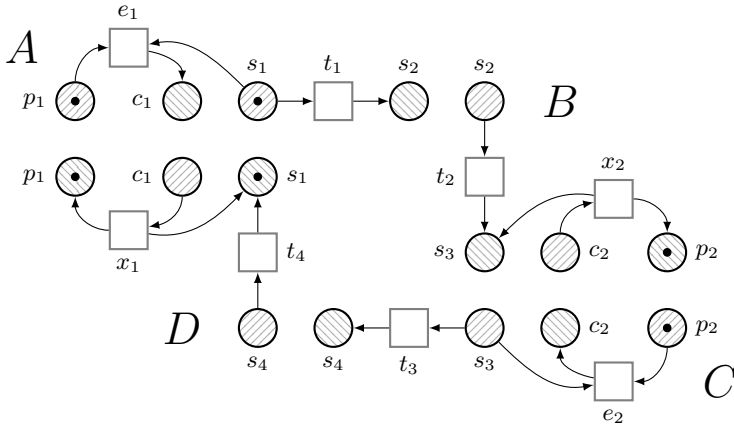
**Size reduction of components.** If the components are not as small as those in Fig. 1 we might like to split them up even more as the number of siphons and traps of a component may grow exponentially with its size, i.e. the number of places. Two transitions with a common place in either their presets or in their postsets always belong to the same component. To force them to different components we need to split up the place *before* we dissolve the net into its components. We propose the following operation, which will replace one place by a circle of alternating places and transitions.

**Definition 6 (Replacing places).** Let  $N = (S, T, F, m_0, S_i, S_o)$  be an open net and  $p$  an inner place of  $S$ . Take any partition  $P = \{T_i \mid 1 \leq i \leq n\}$  of  $\bullet p \cup p \bullet$  where  $n$  is the number of the sets  $T_i$  in  $P$ . We define  $N(p, P) := (S', T', F', m'_0, S_i, S_o)$  by the following algorithm:

- Start with  $S' = S \setminus \{p\}$ ,  $T' = T$ ,  $F' = F|_{(S' \times T) \cup (T \times S')}$  and  $m'_0 = m_0|_{S'}$ .
- For each  $T_i$  add a place  $p_i$  and for each  $t \in T_i$  connect it like  $p$ :  $F'(p_i, t) = F(p, t)$  and  $F'(t, p_i) = F(t, p)$ .
- If exists  $t \in T_i$  with  $F'(t, p_i) > 0$  add  $p_i^e$  and  $t_i^e$  with  $F'(p_i^e, t_i^e) = F'(t_i^e, p_i) = 1$ .
- If exists  $t \in T_i$  with  $F'(p_i, t) > 0$  add  $p_i^x$  and  $t_i^x$  with  $F'(p_i, t_i^x) = F'(t_i^x, p_i^x) = 1$ .
- For each  $i \in \{1, \dots, n\}$  identify the last existing place of the list  $p_i^e, p_i, p_i^x$  with the first one of the list  $p_{(i \bmod n)+1}^e, p_{(i \bmod n)+1}, p_{(i \bmod n)+1}^x$ , forming a circle of all the newly added places and transitions.



**Fig. 2.** A semaphore net  $N$ . For the two processes  $p_1$  and  $p_2$  on the left and right, transitions  $e$  and  $x$  mean entry to and exit from the critical section  $c$ , the semaphore is place  $s$ . Note that  $N$  has an empty interface.



**Fig. 3.** The semaphore place  $s$  has been replaced by a circle (consisting of the  $s_i$  and  $t_i$ ). The semaphore net dissolves into four components  $A$ ,  $B$ ,  $C$ , and  $D$ , where places to be identified when rejoining the components have been given the same label.

- Set  $m'_0(p_1) = m_0(p)$  and  $m'_0(s) = 0$  for all other places on the newly formed circle.

*Example.* Consider the semaphore net of Fig. 2 with the two processes  $p_1$ - $e_1$ - $c_1$ - $x_1$  and  $p_2$ - $e_2$ - $c_2$ - $x_2$  being in their critical section at  $c_1$  and  $c_2$ , respectively, and the semaphore place  $s$ . The net only has two components, one with the transitions  $e_1$  and  $e_2$ , the other with  $x_1$  and  $x_2$ . We cannot split it along the process boundaries, as both processes need read and write access to the place  $s$ .

If we replace  $s$  by a circle of four places and transitions, we obtain four components  $A$ ,  $B$ ,  $C$ , and  $D$  as shown in Fig. 3. It becomes possible now to merge components such that we get subnets  $A \oplus D$  and  $B \oplus C$  consisting of one full process each. These compositions have the smallest number of traps and siphons of all combinations of two components, which reduces time and space needed for the conquer part of our algorithm. Accidentally (or not), these are also the compositions with the smallest interfaces.  $\square$

The question is now what will happen to the traps and siphons if we replace a place by a complete circle. We find the nice property that traps and siphons are bijectively mapped between the two nets.

**Proposition 4 (Unchanged traps and siphons).** *Let  $N$  be an open net with an inner place  $p$  and  $P$  a partition of  $\bullet p \cup p \bullet$ . For  $N(p, P)$  according to definition 6 let  $r$  be a map with  $r(p_i^e) = r(p_i) = r(p_i^x) = p$  for all places added in the construction of  $N(p, P)$  and  $r(s) = s$  for all places  $s$  of  $N$  except  $p$ . Then, for all subsets  $X$  of places of  $N$ :  $X$  is a trap of  $N$  iff  $r^{-1}(X)$  is a trap of  $N(p, P)$  and  $X$  is a siphon of  $N$  iff  $r^{-1}(X)$  is a siphon of  $N(p, P)$ . Furthermore, all traps and siphons of  $N(p, P)$  have the form  $r^{-1}(X)$ .*

*Proof.* We show this for traps only; for siphons the proposition then follows from symmetry. Let  $r^{-1}(X) \in \mathcal{Q}(N(p, P))$  and  $t \in X \bullet$  a transition of  $N$ . Then,  $t$  is also a transition in  $N(p, P)$  and  $t \in r^{-1}(X) \bullet$  by Def. 6. As  $r^{-1}(X)$  is a trap,  $t \in \bullet r^{-1}(X)$  in  $N(p, P)$  and therefore also  $t \in \bullet X$  in  $N$ . We conclude  $X \in \mathcal{Q}(N)$ . The same argument holds for  $X \in \mathcal{Q}(N)$  and  $t \in r^{-1}(X) \bullet$  in  $N(p, P)$  if we just swap  $X$  with  $r^{-1}(X)$  and  $N$  with  $N(p, P)$ . We conclude  $r^{-1}(X) \in \mathcal{Q}(N(p, P))$  then.

Let now  $Y$  be a trap of  $N(p, P)$ . If  $Y$  does not contain any of the  $p_i^e/p_i/p_i^x$ , then  $p \notin r(Y)$  and  $r$  is the identity on  $Y$ . We conclude  $Y = r^{-1}(r(Y))$ . If  $Y$  contains at least one of the  $p_i^e/p_i/p_i^x$  we get  $p \in r(Y)$ . By the trap property, if a  $t_i^e \in p_i^e \bullet$  for some  $p_i^e \in Y$ , also  $t_i^e \in \bullet Y$  must hold, i.e.  $p_i \in Y$ . Analogously, for  $t_i^x \in p_i \bullet$  with  $p_i \in Y$  also  $t_i^x \bullet = \{p_i^x\} \subseteq Y$  holds. In any case, if one of the  $p_i^e/p_i/p_i^x$  belongs to  $Y$ , all of them do for all  $i$ . So again,  $Y = r^{-1}(r(Y))$ .  $\square$

Note that Def. 6 cannot be applied to interface places. This would change the number of siphons and traps in the net, as the circle constructed in the definition cannot contain interface places. Logically, the best time to apply Def. 6 is then at the beginning, when we usually have a closed net and could replace *all* the places. Then, components would all look like the one depicted in Fig. 4, where for each place in the preset or postset of the main transition  $t$  one link of the corresponding circle created by Def. 6 is added.

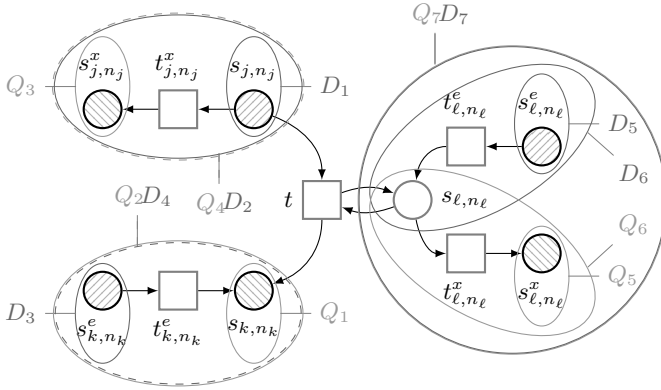
Not all of the sets of traps and siphons in figure 4 need to be considered for our divide-and-conquer approach, since Prop. 4 tells us that the circles of Def. 6 appear either completely or not at all in any trap or siphon of the whole net. That means, only the sets  $Q_2, D_2, Q_7, D_7, Q_4 \times (Q_2 \cup Q_7)$ , and  $D_4 \times (D_2 \cup D_7)$  and unions of two or more traps or two or more siphons from these sets will be relevant subsets of traps and siphons of the overall net.

### 4.2 Composing Siphons and Traps

There is a good reason for using open net decomposition rather than any other style of decomposition.

**Lemma 2.** *Let  $N_1$  and  $N_2$  be open nets with  $N_k = (S_k, T_k, F_k, m_{0,k}, S_{i,k}, S_{o,k})$  for  $k = 1, 2$ .*

- *If  $D$  is a siphon (or trap, resp.) in  $N_1 \oplus N_2$  then  $D \cap S_1$  is either empty or a siphon (or trap, resp.) in  $N_1$  and  $D \cap S_2$  is either empty or a siphon (or trap, resp.) in  $N_2$ .*



**Fig. 4.** A component  $C(t)$  for some transition  $t$  with  $\bullet t = \{s_j, s_\ell\}$  and  $t^\bullet = \{s_k, s_\ell\}$  as given by definition 6. Ellipses show the traps and siphons. The dashed ellipses  $Q_4$  and  $D_4$  are not traps or siphons (due to  $t$ ) and need to be unified with traps from  $Q_1/Q_2/Q_6/Q_7$  and siphons from  $D_1/D_2/D_6/D_7$  first, respectively.

- If  $D_1$  is a siphon (or trap, resp.) in  $N_1$  and  $D_2$  is a siphon (or trap, resp.) in  $N_2$  such that  $D_1 \cap S_2 = D_2 \cap S_1$  (i.e. their interfaces to the respective other component are equal) then  $D_1 \cup D_2$  is a siphon (or trap, resp.) in  $N_1 \oplus N_2$ .

*Proof.* This follows easily from the constraints on interface places in open nets. Empty sets occur if  $D$  lies completely in the inner part of either  $N_1$  or  $N_2$ .  $\square$

*Example.* In Fig. 11  $\{a, d, e\}$  is a siphon and a trap of  $N_1 \oplus N_2$ . It decomposes into the siphons (and traps)  $\{a, d\}$  of  $N_1$  and  $\{d, e\}$  of  $N_2$ . The other way round,  $\{a, c\}$  is a siphon of  $N_1$ ,  $\{c\}$  is a siphon of  $N_2$ .  $c$  and  $d$  are the shared places of the interfaces of  $N_1$  and  $N_2$ . Hence  $\{a, c\}$  is a siphon in  $N_1 \oplus N_2$ .  $\square$

From Lemma 2, the general idea of our approach is obvious. We collect, for each part of the interface of a component, the open siphons and the included traps, together with their interface. Upon composition, we merge siphons and traps with equal interface. Unfortunately, given an interface with  $k$  places, there are  $2^k$  potential interfaces for siphons to be considered, and for each siphon, a contained trap can have an interface that spans over any subset of the interface of the siphon. Consequently, we need to further investigate regularities that arise from the open net shape of the components. To this end, we shall heavily exploit the following simple observations on siphons and traps.

**Proposition 5 (Properties of Siphons)**

- (1) The union of siphons is a siphon.
- (2) Let  $D$  be a siphon and  $X \subseteq D$ . There is an  $X$ -minimal siphon  $D' \subseteq D$ .
- (3) Let  $D$  be a  $\emptyset$ -minimal siphon in  $N_1 \oplus N_2$ . Then, if not empty,  $D \cap S_1$  is a  $(D \cap S_1 \cap S_2)$ -minimal siphon in  $N_1$  and  $D \cap S_2$  is a  $(D \cap S_1 \cap S_2)$ -minimal siphon in  $N_2$ .

The same observations hold for traps.

*Proof.* (1) and (2) are trivial. For (3) the places  $D \cap S_1 \cap S_2$  are forced in the siphons while the remaining places follow by the same reasoning as for  $D$  in  $N_1 \oplus N_2$ , i.e. the structure of the net.

Let us first reduce the number of siphons to be considered. Consider two components  $N_1$  and  $N_2$  and a set of shared places  $X \subseteq S_1 \cap S_2$ . By Lemma 2, for every pair of siphons  $D_1$  of  $N_1$  and  $D_2$  of  $N_2$  where  $D_1 \cap S_2 = X = D_2 \cap S_1$ ,  $D_1 \cup D_2$  is a siphon in  $N_1 \oplus N_2$ . However, some of these siphons may contain more or better (i.e. marked) traps than others.

**Definition 7 (Worse siphons).** Let  $N$  be an open net and  $X \subseteq S_i \cup S_o$ . Let  $D_1$  and  $D_2$  be siphons with  $D_1 \cap (S_i \cup S_o) = X = D_2 \cap (S_i \cup S_o)$ . Call  $D_1$  worse than  $D_2$  iff, for every  $Y \subseteq X$ ,

- If  $D_1$  contains a trap  $Q_1$  with  $Q_1 \cap X = Y$  then  $D_2$  contains a trap  $Q_2$  with  $Q_2 \cap X = Y$ .
- If  $D_1$  contains a marked trap  $Q_1$  with  $Q_1 \cap X = Y$  then  $D_2$  contains a marked trap  $Q_2$  with  $Q_2 \cap X = Y$ .

*Example.* In  $N_1 \oplus N_2$  of Fig. 1, siphons  $\{a, c, e\}$  and  $\{a, d, e\}$  are mutually worse than each other, so only one of them has to be considered in larger compositions. Assuming a token on  $d$ ,  $\{a, c, e\}$  is worse than  $\{a, d, e\}$  but not vice versa. □

**Lemma 3.** Let  $N_1$  and  $N_2$  be open nets. Let  $D_1$  be a siphon of  $N_1$  and let  $D_2$  be a siphon of  $N_2$  such that  $D_1 \cap S_2 = D_2 \cap S_1$ . Let  $D_1$  be worse than  $D'_1$  and  $D_2$  be worse than  $D'_2$ . Then  $D_1 \cup D_2$  is worse than  $D'_1 \cup D'_2$  in  $N_1 \oplus N_2$ .

In particular, if the union of the worse siphons includes a marked trap, so does the union of the better siphons. Consequently, we may remove a siphon from any consideration in a component as long as we keep a worse one. Although *worse than* is only a preorder and no partial order, we shall sloppily refer to the *worst* siphons as a (as small as possible) set of siphons that needs to be kept according to Lemma 3. The next observation rephrases the well-known fact that it is sufficient to check  $\emptyset$ -minimal siphons for evaluating the STP.

**Corollary 1.** Let  $D_1$  and  $D_2$  be siphons of an open net  $N$  with  $D_1 \cap (S_i \cup S_o) = D_2 \cap (S_i \cup S_o)$ . If  $D_1 \subseteq D_2$  then  $D_1$  is worse than  $D_2$ .

While the previous result reduces the number of siphons to be considered for a given interface, the following investigations concern the number of different interfaces to be explicitly considered. We shall argue, that finally we only need to consider *elementary* siphons and traps.

**Definition 8 (Elementary siphons and traps).** A siphon  $D$  of an open net  $N$  is elementary iff there is a place  $s \in S_i \cup S_o$  such that  $D$  is  $\{s\}$ -minimal. A trap  $Q$  is interface-elementary iff there is a place  $s \in S_i \cup S_o$  such that  $Q$  is  $\{s\}$ -minimal.  $Q$  is token-elementary iff there is a place  $s$  where  $m_0(s) > 0$  and  $Q$  is  $\{s\}$ -minimal.

*Example.* In the open net  $N_1$  of Fig. [11](#),  $\{a\}$ ,  $\{b\}$ ,  $\{a, c\}$ ,  $\{a, d\}$ ,  $\{b, c\}$ ,  $\{b, d\}$  are the elementary siphons. Although  $\{a\}$  is included in  $\{a, c\}$ , we want to keep both as  $\{a, c\}$  is  $\{c\}$ -minimal while  $\{a\}$  is not. The interface-elementary traps are  $\{a, c\}$ ,  $\{a, d\}$ ,  $\{b, c\}$ ,  $\{b, d\}$ ,  $\{c\}$ ,  $\{d\}$ . There are no token-elementary traps. Assuming a token on  $a$ ,  $\{a, c\}$ ,  $\{a, d\}$  would become token-elementary. Assuming instead a token on  $c$ , the only token-elementary trap would be  $\{c\}$ . In particular, the definition states that  $\{b, c\}$  is not a token-elementary trap.  $\square$

Note that a token-elementary trap may be closed (i.e. disjoint to the interface) while an interface-elementary trap is always open. The following facts justify this selection.

**Lemma 4.** *Let  $N$  be an open net.*

- (1) *For every open siphon  $D$  of  $N$  there is a worse union of elementary siphons.*
- (2) *If a siphon  $D$  contains a trap  $Q$  then it contains some union of interface-elementary traps  $Q_1 \cup \dots \cup Q_k$  where  $Q \cap (S_i \cup S_o) = (Q_1 \cup \dots \cup Q_k) \cap (S_i \cup S_o)$ .*
- (3) *If a siphon  $D$  contains a marked trap  $Q$  then it contains some union of traps  $Q_1 \cup \dots \cup Q_k \cup Q_m$  where  $Q \cap (S_i \cup S_o) = (Q_1 \cup \dots \cup Q_k \cup Q_m) \cap (S_i \cup S_o)$ ,  $Q_1, \dots, Q_k$  are interface-elementary, and  $Q_m$  is token-elementary.*

*Proof.* (1) Let  $X = D \cap (S_i \cup S_o) \neq \emptyset$ . For each  $s \in X$ , let  $D_s$  be an  $\{s\}$ -elementary siphon included in  $D$ . Obviously,  $\bigcup_{s \in X} D_s$  has the same interface as  $D$ , is contained in  $D$ , and not empty. By Cor. [11](#), it is worse than  $D$ . Claims (2) and (3) can be proven analogously, but note that  $k = 0$  holds in the unions if  $Q$  is closed.  $\square$

In consequence, we only need to store information about elementary siphons, elementary traps, and information about inclusion of elementary traps in unions of elementary siphons. The advantage of using elementary traps and siphons is their simple structure. The following is trivial.

**Lemma 5.** *Let  $N$  be an open net.*

- (1) *For  $s \in S_i$ ,  $\{s\}$  is the only  $\{s\}$ -minimal siphon of  $N$ . For  $s \in S_o$ ,  $\{s\}$  is the only  $\{s\}$ -minimal trap of  $N$ .*
- (2) *For  $s \in S_o$  and an  $\{s\}$ -minimal siphon  $D$ ,  $D \cap S_o = \{s\}$ . For  $s \in S_i$  and an  $\{s\}$ -minimal trap  $Q$ ,  $Q \cap S_i = \{s\}$ .*

**Definition 9 (Wrapping siphons).** *A family  $\mathcal{M} = \{D_1, \dots, D_k\}$  of sets of places wraps a set  $Q$  of places iff  $Q \subseteq D_1 \cup \dots \cup D_k$  and this is not the case for any proper subset of  $\mathcal{M}$ .*

*Example.* In the net  $N_1 \oplus N_2$  of Fig. [11](#), the family of siphons  $\{\{a, c\}, \{b, d, e\}\}$  wraps the trap  $\{c, e\}$ .  $\square$

*Remark 1.* Let  $\mathcal{M}$  be a family of elementary siphons. The union of  $\mathcal{M}$  includes a trap  $Q$  if and only if  $Q$  is wrapped by some subset of  $\mathcal{M}$ .

Even among the elementary siphons, some siphon  $D$  may be redundant. This is the case if, for all siphons that can be constructed using  $D$ , a worse one can be constructed without using  $D$ .

**Definition 10 (Redundant elementary siphon).** *Let  $N$  be an open net and  $\mathcal{M}$  a set of elementary siphons. Siphon  $D \in \mathcal{M}$  is redundant iff, for all  $\mathcal{M}_1 \subseteq \mathcal{M}$  there exists another subset  $\mathcal{M}_2 \subseteq \mathcal{M}$  where  $\bigcup(\mathcal{M}_2 \setminus \{D\})$  is worse than  $\bigcup(\mathcal{M}_1 \cup \{D\})$ . ( $\bigcup X$  without a subscript stands for  $\bigcup_{x \in X} x$ .)*

*Example.* In  $N_1 \oplus N_2$  of Fig. 1, any of the elementary siphons  $\{a, c, e\}$  and  $\{a, d, e\}$  is redundant. In fact, any interface constellation of siphons and traps that can be composed from elementary objects and  $\{a, c, e\}$  can as well be generated using  $\{a, d, e\}$ . After removing one of them, the other one is no longer redundant as it is then the only one remaining with interface  $\{a, e\}$ . If we put a token on  $d$ , only  $\{a, d, e\}$  is redundant. For any constellation of siphons and included traps that can be constructed using  $\{a, d, e\}$ , a worse one (particularly with some unmarked traps instead of marked traps) can be generated using  $\{a, c, e\}$ .  $\square$

### 4.3 Representing Information about Open Siphons and Traps

From the considerations of the previous subsection, we conclude that we need to provide the following information about an open net.

**Definition 11 (Information about components).** *Let  $N$  be an open net,  $\mathcal{M}_D$  a set of elementary siphons that can be obtained from the set of all elementary siphons by removing (one by one) redundant ones,  $\mathcal{M}_Q$  the set of interface-elementary traps in  $N$ , and  $\mathcal{M}_M$  the set of all token-elementary traps in  $N$ . Fix a set  $\Sigma$  with elements from an arbitrary universe such that  $\text{card}(\Sigma) = \text{card}(\mathcal{M}_D)$  and fix some bijection  $l$  between  $\Sigma$  and  $\mathcal{M}_D$  (elements of  $\Sigma$  serve as names for elementary siphons). We keep track of the following information about  $N$ :*

- The set  $\Sigma$  introducing names for elementary siphons;
- A mapping  $\text{int} : \Sigma \rightarrow \wp(S_i \cup S_o)$ ,  $x \mapsto l(x) \cap (S_i \cup S_o)$  recording the interfaces of the elementary siphons;
- The set  $L_Q = \{Q \cap (S_i \cup S_o) \mid Q \in \mathcal{M}_Q\}$  introducing the interfaces of the interface-elementary traps;
- The set  $L_M = \{Q \cap (S_i \cup S_o) \mid Q \in \mathcal{M}_M\}$  introducing the interfaces of the token-elementary traps;
- The mapping  $w_Q : L_Q \rightarrow \wp(\wp(\Sigma))$ ,  $X \mapsto \{l^{-1}(\mathcal{M}) \mid \exists Q \in \mathcal{M}_Q : Q \cap (S_i \cup S_o) = X, \mathcal{M} \text{ wraps } Q\}$  recording the wrapping sets of elementary siphons for all interface-elementary traps with a given interface;
- The mapping  $w_M : L_M \rightarrow \wp(\wp(\Sigma))$ ,  $X \mapsto \{l^{-1}(\mathcal{M}) \mid \exists Q \in \mathcal{M}_M : Q \cap (S_i \cup S_o) = X, \mathcal{M} \text{ wraps } Q\}$  recording the wrapping sets of elementary siphons for all token-elementary traps with a given interface;

*Example.* The full information about  $N_1$  in Fig. 1 reads as follows.

- $\Sigma_1 = \{1, 2, 3, 4, 5, 6\}$ ;
- $\text{int}_1(1) = \{a\}, \text{int}_1(2) = \{b\}, \text{int}_1(3) = \{a, c\}, \text{int}_1(4) = \{a, d\}, \text{int}_1(5) = \{b, c\}, \text{int}_1(6) = \{b, d\}$ ;
- $L_{Q1} = \{\{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c\}, \{d\}\}$ ;
- $L_{M1} = \emptyset$ ;



- $w_{Q1}(\{a, c\}) = \{\{3\}\}, w_{Q1}(\{a, d\}) = \{\{4\}\}, w_{Q1}(\{b, c\}) = \{\{5\}\},$   
 $w_{Q1}(\{b, d\}) = \{\{6\}\}, w_{Q1}(\{c\}) = \{\{3\}, \{5\}\}, w_{Q1}(\{d\}) = \{\{4\}, \{6\}\};$
- $w_M = \emptyset.$

Assuming a token on  $c$ , we would obtain  $L_M = \{\{c\}\}$  and  $w_M(\{c\}) = \{\{3\}, \{5\}\}$ . With a token on  $b$  instead, we would get  $L_M = \{\{b, c\}, \{b, d\}\}, w_M(\{b, c\}) = \{\{5\}\},$  and  $w_M(\{b, d\}) = \{\{6\}\}$ . For later use, we provide the full information for  $N_2$  although it does not provide new insights.

- $\Sigma_2 = \{7, 8, 9, 10, 11, 12\};$
- $int_2(7) = \{c\}, int_2(8) = \{d\}, int_2(9) = \{c, e\}, int_2(10) = \{c, f\}, int_2(11) = \{d, e\}, int_2(12) = \{d, f\};$
- $L_{Q2} = \{\{c, e\}, \{c, f\}, \{d, e\}, \{d, f\}, \{e\}, \{f\}\};$
- $L_{M2} = \emptyset;$
- $w_{Q2}(\{c, e\}) = \{\{9\}\}, w_{Q2}(\{c, f\}) = \{\{10\}\}, w_{Q2}(\{d, e\}) = \{\{11\}\},$   
 $w_{Q2}(\{d, f\}) = \{\{12\}\}, w_{Q2}(\{e\}) = \{\{9\}, \{11\}\}, w_{Q2}(\{f\}) = \{\{10\}, \{12\}\};$
- $w_M = \emptyset.$  □

In the remainder of this section, we argue that this information for some open nets  $N_1$  and  $N_2$  is sufficient for reasoning about siphons and traps of  $N_1 \oplus N_2$ . Let us first consider closed siphons in  $N_1 \oplus N_2$ . If a closed siphon is already a closed one in either  $N_1$  or  $N_2$ , we assume that this siphon has been checked for elementary components, or has been checked during an earlier composition step. It is thus sufficient to consider those siphons  $D$  that spread over both  $N_1$  and  $N_2$ . By the considerations in the previous subsection, it is sufficient to check those siphons for included marked traps which can be composed by elements of  $\mathcal{M}_D$ . Concerning the included traps, it is sufficient to check traps that can be composed by elements of  $\mathcal{M}_Q$  and a single element of  $\mathcal{M}_M$ . We propose to execute the necessary checks simultaneously for all siphons by translating the check into a Boolean formula. The formula is satisfied if and only if some siphon of  $N_1 \oplus N_2$  that spreads over both components does not contain a marked trap. The propositions of the formula are elements of  $\Sigma_1$  and  $\Sigma_2$ , i.e. the symbols representing the elementary siphons of the two components (which we silently assume to be disjoint). The satisfying assignment assigns true to the names of those elementary siphons whose composition is a siphon that proves STP not to hold.

The formula consists of three parts. In the first part, we state that the represented siphon is not empty. In the second part, we state that the projections of the siphon to the components generate the same interface. In the third part, we state that the composition does not include a marked trap. The trick for stating the third part is to state that the siphon represented by the satisfying assignment does not include any wrap for at least one elementary trap participating in a trap of the composed system. Traps in the composed system are formed by a union of traps of the components such that the union of elementary traps in  $N_1$  have the same interface to  $N_2$  which the union of elementary traps of  $N_2$  has to  $N_1$ . The following definition boils this idea down to interface considerations. As the same technique is later on needed for siphons as well, we already present matching for siphons as well.

**Definition 12 (Matching).** Let  $N_1$  and  $N_2$  be open nets with information attached according to Def. [17](#). A token trap matching is a tuple  $[X_1, Y_1, X_2, Y_2]$  such that  $X_1 \subseteq L_{M_1}$ ,  $Y_1 \subseteq L_{Q_1}$ ,  $X_2 \subseteq L_{M_2}$ ,  $Y_2 \subseteq L_{Q_2}$ ,  $\text{card}(X_1) + \text{card}(X_2) = 1$ ,  $\bigcup(X_1 \cup Y_1) = \bigcup(X_2 \cup Y_2)$ . An interface trap matching is a tuple  $[Y_1, Y_2]$  such that  $Y_1 \subseteq L_{Q_1}$ ,  $Y_2 \subseteq L_{Q_2}$ ,  $\bigcup Y_1 = \bigcup Y_2$ . A siphon matching is a tuple  $[Z_1, Z_2]$  such that  $Z_1 \subseteq \Sigma_1$ ,  $Z_2 \subseteq \Sigma_2$ , and  $\bigcup_{\sigma_1 \in Z_1} \text{int}(\sigma_1) = \bigcup_{\sigma_2 \in Z_2} \text{int}(\sigma_2)$ . A matching is minimal iff no different matching is pointwise set-included. A token trap matching is internal iff  $\bigcup X_1 \cup \bigcup Y_1 \subseteq S_2$  and  $\bigcup X_2 \cup \bigcup Y_2 \subseteq S_1$ . The interface of a trap matching is  $(\bigcup(X_1 \cup Y_1 \cup X_2 \cup Y_2)) \setminus (S_1 \cap S_2)$  resp.  $(\bigcup(Y_1 \cup Y_2)) \setminus (S_1 \cap S_2)$ .

*Example.* There are no token-minimal matchings for  $N_1$  and  $N_2$  in Fig. [11](#). Examples of minimal siphon matchings are  $[\{1\}, \emptyset]$ ,  $[\{3\}, \{7\}]$ , or  $[\{3\}, \{9\}]$ . Examples for minimal trap matchings are  $[\emptyset, \{\{e\}\}]$  or  $[\{\{c\}\}, \{\{c, e\}\}]$ . Assuming a token on  $c$  in both components,  $[\{\{c\}\}, \emptyset, \emptyset, \{\{c, e\}\}]$  would be a minimal token trap matching.  $\square$

Minimal matchings can be easily determined by a saturation algorithm. Start with an individual element. That may lead to interface places  $s$  that are not in the respective other open net. Add (nondeterministically) an  $\{s\}$ -minimal object of the other component and proceed until all interface places are matched. If there is no  $\{s\}$ -minimal object, just backtrack.

The definition shows that a token trap matching represents the union of those elementary traps that form a smallest marked trap in  $N_1 \oplus N_2$ . A trap which is fully contained in one of the components and does not touch interface places leads to a trap matching where one  $X_i$  is non-empty while both  $Y_i$  are empty.

**Definition 13 (Formula assigned to  $N_1$  and  $N_2$ ).** Let  $N_1$  and  $N_2$  be open nets. Then the corresponding formula  $\phi(N_1, N_2)$  is built as follows:

$$\phi(N_1, N_2) = \phi_1 \wedge \phi_2 \wedge \phi_3$$

where, for  $i \in \{1, 2\}$ ,  $\Sigma'_i = \{\sigma \mid \sigma \in \Sigma_i, \text{int}(\sigma) \subseteq S_{2-i}\}$  and

$$\begin{aligned} \phi_1 &= \bigvee_{x \in \Sigma'_1 \cup \Sigma'_2} x \\ \phi_2 &= \bigwedge_{x \in \Sigma'_1} (x \implies \bigwedge_{s \in \text{int}(x) \cap S_{1,1}} \bigvee_{y \in \Sigma'_2: s \in \text{int}(y)} y) \wedge \\ &\quad \bigwedge_{x \in \Sigma'_2} (x \implies \bigwedge_{s \in \text{int}(x) \cap S_{1,2}} \bigvee_{y \in \Sigma'_1: s \in \text{int}(y)} y) \\ \phi_3 &= \bigwedge_{[X_1, Y_1, X_2, Y_2] \text{ is internal minimal token trap matching}} \\ &\quad (\bigvee_{N \in X_1} \bigwedge_{\Sigma^* \in w_{M_1}(N)} \bigvee_{\sigma \in \Sigma^*} \neg \sigma \vee \\ &\quad \bigvee_{N \in Y_1} \bigwedge_{\Sigma^* \in w_{Q_1}(N)} \bigvee_{\sigma \in \Sigma^*} \neg \sigma \vee \\ &\quad \bigvee_{N \in X_2} \bigwedge_{\Sigma^* \in w_{M_2}(N)} \bigvee_{\sigma \in \Sigma^*} \neg \sigma \vee \\ &\quad \bigvee_{N \in Y_2} \bigwedge_{\Sigma^* \in w_{Q_2}(N)} \bigvee_{\sigma \in \Sigma^*} \neg \sigma) \end{aligned}$$

*Example.* For the composition of  $N_1$  and  $N_2$  in Fig. [11](#), we obtain  $\Sigma'_1 = \emptyset$  and  $\Sigma'_2 = \{7, 8\}$ , so any assignment satisfying  $\phi_1$  ensures that the second part of  $\phi_2$  and therefore  $\phi_2$  overall will be false. Informally this means that all siphons in  $N_1 \oplus N_2$  touch the interface of  $N_1 \oplus N_2$  so nothing needs to be checked. For obtaining a nontrivial formula, rename  $e$  to  $a$  and  $f$  to  $b$  in Fig. [11](#). In that case, we obtain

- $\phi_1 = 1 \vee \dots \vee 6 \vee 7 \vee \dots \vee 12$ ;
- $\phi_2 = (1 \implies (9 \vee 11)) \wedge (2 \implies (10 \vee 12)) \wedge (3 \implies (9 \vee 11)) \wedge (4 \implies (9 \vee 11)) \wedge \dots \wedge (12 \implies (4 \vee 6))$ ;
- $\phi_3 = true$

$\phi_3$  is true as there are no tokens in the system and the empty conjunction is always true. This leads to satisfying assignments. For instance, assigning true to 3 and 9 would satisfy the whole formula. Indeed, the represented siphon  $\{a, c, e\}$  does not contain a marked trap.

Assuming a token on  $c$  in both components, we would need to include formulas for each internal minimal token trap matching. An example for such a matching is  $\{\{\{c\}\}, \{\{a, c\}\}, \emptyset, \{\{c, e = a\}\}\}$ . This matching would contribute the following subformula to  $\phi_3 = (\neg 3 \wedge \neg 5) \vee \neg 3 \vee \neg 9$ . This subformula states that the trap  $\{a, c, e = a\}$  be not included in any siphon represented by a satisfying assignment of the formula.  $\square$

**Theorem 2.** *Let  $N_1$  and  $N_2$  be open nets.  $\phi(N_1, N_2)$  is satisfiable if and only if there exists a siphon  $D$  of  $N_1 \oplus N_2$  such that  $D \cap S_1 \cap S_2 \neq \emptyset$  and  $D$  does not contain any marked trap.*

*Proof.* ( $\rightarrow$ ) Let  $\beta$  be a satisfying assignment of  $\phi(N_1, N_2)$  and consider the set of places  $D_1 \cup D_2$  with  $D_1 = \bigcup_{\sigma \in \Sigma_1: \beta(\sigma) = true} l_1(\sigma)$  and  $D_2 = \bigcup_{\sigma \in \Sigma_2: \beta(\sigma) = true} l_2(\sigma)$ . Here,  $l_i$  are the mappings used in Def. 12 for  $N_i$ , resp. As we composed elementary siphons,  $D_1$  is a siphon of  $N_1$  and  $D_2$  is a siphon of  $N_2$ . By  $\phi_2$ , both siphons share the same interface places,  $D_1 \cup D_2$  is a siphon of  $N_1 \oplus N_2$ .  $\phi_1$  tells us that this siphon is not empty since it contains at least one elementary siphon and elementary siphons cannot be empty. Assume  $D_1 \cup D_2$  contains a marked trap. By Lemma 4, it also contains a union of some interface-elementary and one token-elementary trap of  $N_1$  or  $N_2$  or both. A minimal such union defines a token trap matching for which a corresponding subformula is part of  $\phi_3$ . This subformula asserts that for at least one trap participating in the considered union of elementary traps (second level operator),  $D_1 \cup D_2$  does not contain sufficiently many elementary siphons to include that elementary trap. In consequence, the whole trap cannot be contained in  $D_1 \cup D_2$ .

( $\leftarrow$ ) Assume there is a siphon  $D$  in  $N_1 \oplus N_2$  that contains places in  $S_1 \cap S_2$  and does not contain a marked trap. By Lemma 4,  $D$  includes a siphon  $D'$  that is the union of elementary siphons and which is obviously unmarked as well. Since we only leave out redundant elementary siphons in Def. 11, a siphon  $D''$  can be constructed from the elementary siphons in  $\mathcal{M}_{D_1}$  and  $\mathcal{M}_{D_2}$  such that  $D'' \cap S_1$  is worse than  $D' \cap S_1$  and  $D'' \cap S_2$  is worse than  $D' \cap S_2$ . By Def. 7,  $D''$  cannot contain a marked trap either. Consider the assignment  $\beta$  that assigns true to all symbols that represent elementary siphons participating in  $D''$ . As  $D''$  has the same (non-empty) set of places in  $S_1 \cap S_2$ ,  $D''$  is not empty. Consequently,  $D''$  includes at least one elementary siphon and thus  $\phi_1$  must be satisfied. Further,  $D'' \cap S_1$  and  $D'' \cap S_2$  share the same places in  $S_1 \cap S_2$ , so  $\phi_2$  must be satisfied. Finally, since  $D''$  does not contain a marked trap, no union of a subset of the used elementary siphons wraps a marked trap. Thus, each wrap of any marked

trap must contain one siphon that is not used to form  $D''$ . Consequently,  $\phi_3$  is satisfied.  $\square$

Let us now shift our attention to the open siphons of  $N_1 \oplus N_2$ . We need to produce the information (according to Def. [11](#)) for  $N_1 \oplus N_2$  from the information for  $N_1$  and the one for  $N_2$ .

There are two kinds of open siphons and traps in  $N_1 \oplus N_2$ . First there are those fully contained in one of the components, i.e. disjoint to either  $S_1$  or  $S_2$ . They are elementary if and only if they are elementary in their component, and they are wrapped by elements of their own component only. They can be recognised by having no interface places in common with the set of places of the other component. Consequently, information about these siphons and traps can be directly copied from the information provided by the respective component.

Second, there are siphons and traps that spread over both components. Such a siphon (or trap, resp.) is composed of a set of elementary siphons (traps, resp.) of both components. We only need to consider such a siphon if it also contains places in  $S_i \cup S_o$  since otherwise it can be decomposed into disjoint siphons (or traps) of the individual components. Thus, the strategy of composing elementary siphons and traps of the components to siphons and traps of  $N_1 \oplus N_2$  is to find the smallest sets of individual siphons and traps of  $N_1$  and  $N_2$  that match at  $S_1 \cap S_2$ . A composite trap is wrapped by a set of siphons if and only if each individual elementary trap is wrapped within its own component and the resulting set of siphons is minimal. All the described information can be computed from the abstracted information that is provided by the components.

**Definition 14 (Information for  $N_1 \oplus N_2$ ).** *Let, for  $i \in \{1, 2\}$ ,  $[\Sigma_i, \text{int}_i, L_{Q_i}, L_{M_i}, w_{Q_i}, w_{M_i}]$  be the information for  $N_i$ . Define the information for  $N_1 \oplus N_2$  as  $[\Sigma, \text{int}, L_Q, L_M, w_Q, w_M]$  with*

- $\Sigma$  be the set of minimal siphon matchings between  $N_1$  and  $N_2$ ;
- for each  $[Z_1, Z_2] \in \Sigma$ , let  $\text{int}([Z_1, Z_2]) = (\bigcup_{\sigma_1 \in Z_1} \text{int}(\sigma_1) \cup \bigcup_{\sigma_2 \in Z_2} \text{int}(\sigma_2)) \setminus (S_1 \cap S_2)$ ;
- $L_Q = \{(\bigcup Y_1 \cup \bigcup Y_2) \setminus (S_1 \cap S_2) \mid [Y_1, Y_2] \text{ is interface trap matching}\}$ ;
- $L_M = \{(\bigcup X_1 \cup \bigcup X_2 \cup \bigcup Y_1 \cup \bigcup Y_2) \setminus (S_1 \cap S_2) \mid [X_1, Y_1, X_2, Y_2] \text{ is token trap matching}\}$ ;
- $w_Q(X) = \{ \{ [Z_{11}, Z_{21}], \dots, [Z_{1k}, Z_{2k}] \} \subseteq \Sigma \mid \text{exists minimal interface trap matching } [Y_1, Y_2] \text{ s.t. } (\bigcup Y_1 \cup \bigcup Y_2) \setminus (S_1 \cap S_2) = X, \text{ and } \forall X' \in Y_1 \exists M \in w_Q(X'): M \subseteq \bigcup_{i=1}^k Z_{1i}, \forall X' \in Y_2 \exists M \in w_Q(X'): M \subseteq \bigcup_{i=1}^k Z_{2i} \}$ ;
- $w_M(X) = \{ \{ [Z_{11}, Z_{21}], \dots, [Z_{1k}, Z_{2k}] \} \subseteq \Sigma \mid \text{exists minimal token trap matching } [X_1, Y_1, X_2, Y_2] \text{ s.t. } (\bigcup Y_1 \cup \bigcup Y_2 \cup \bigcup X_1 \cup \bigcup X_2) \setminus (S_1 \cap S_2) = X, \forall X' \in Y_1 \exists M \in w_Q(X'): M \subseteq \bigcup_{i=1}^k Z_{1i}, \forall X' \in Y_2 \exists M \in w_Q(X'): M \subseteq \bigcup_{i=1}^k Z_{2i}, \forall X' \in X_1 \exists M \in w_M(X'): M \subseteq \bigcup_{i=1}^k Z_{1i}, \forall X' \in X_2 \exists M \in w_M(X'): M \subseteq \bigcup_{i=1}^k Z_{2i} \}$

Within the values of  $w_Q$  and  $w_M$ , we silently assume that supersets of other elements are removed.

*Example.* Let us compose  $N_1$  with  $N_2$  in Fig. [11](#). We need to consider the following 14 siphon matchings. For convenience, we assign a number to each matching.

$13 = [\{1\}, \emptyset]$ ,  $14 = [\{2\}, \emptyset]$ ,  $15 = [\{3\}, \{7\}]$ ,  $16 = [\{3\}, \{9\}]$ ,  $17 = [\{3\}, \{10\}]$ ,  $18 = [\{4\}, \{8\}]$ ,  $19 = [\{4\}, \{11\}]$ ,  $20 = [\{4\}, \{12\}]$ ,  $21 = [\{5\}, \{7\}]$ ,  $22 = [\{5\}, \{9\}]$ ,  $23 = [\{5\}, \{10\}]$ ,  $24 = [\{6\}, \{8\}]$ ,  $25 = [\{6\}, \{11\}]$ ,  $26 = [\{6\}, \{12\}]$ . We can represent the interfaces  $\{a, e\}$ ,  $\{a, f\}$ ,  $\{b, e\}$ ,  $\{b, f\}$ ,  $\{e\}$ , and  $\{f\}$  with interface trap matchings, so these six sets form  $L_Q$ .  $L_M$  is empty as the components do not provide elementary token traps. For computing the wrapping siphons for  $\{a, e\}$ , we need to consider those trap matchings which generate this interface:  $[\{\{a, c\}\}, \{\{c, e\}\}]$  and  $[\{\{a, d\}\}, \{\{d, e\}\}]$ .  $\{a, c\}$  is wrapped by  $\{3\}$ ,  $\{c, e\}$  is wrapped by  $\{9\}$ ,  $\{a, d\}$  is wrapped by  $\{4\}$ , and  $\{d, e\}$  is wrapped by  $\{11\}$ . Hence, we need to look into those siphon matchings which contain any of the siphons 3, 4, 9, or 11. Siphon 3 is contained in 15, 16, and 17. Siphon 4 is contained in 18, 19, and 20. In the second component, siphon 9 is contained in 16 and 22. Siphon 11 is contained in 19 and 25. These siphons need to be combined in a minimal way such that either 3 and 9 or 4 and 11 are contained. Hence, we result in  $w_Q(\{a, e\}) = \{\{15, 22\}, \{16\}, \{17, 22\}, \{18, 25\}, \{19\}, \{20, 25\}\}$ . The remaining values of  $w_Q$  can be computed similarly.  $w_M$  is empty in the example but the principal approach resembles the one for  $w_Q$ .  $\square$

**Theorem 3.** *Let  $N_1$  and  $N_2$  be open nets. Then the information for  $N_1 \oplus N_2$  using Def. 14 is equivalent to the information for  $N_1 \oplus N_2$  according to Def. 17.*

*Proof.* It is easy to see that the definition implements the considerations on siphons and traps of the composed system.  $\square$

The construction of Def. 14 may introduce redundant information. The conditions of Def. 10 can, however, be evaluated by the information available for  $N_1 \oplus N_2$ , so information about redundant elementary siphons can be removed after having applied Def. 14.

*Example.* In the calculation of the previous example, siphons 15, 18, 19, 20, 21, 24, 25, and 26 can be removed through redundancy. Let us verify redundancy for siphon 19. We have to exhibit, for every union  $U$  of elementary siphons containing 19, a worse one  $U'$  not containing 19. In the example it is quite obvious, that, for each interface, there are two elementary siphons in the composition: one that is obtained using common interface place  $c$ , and the other obtained using interface place  $d$ . Call these siphons *dual* to each other. The dual to 19 is 16. For a composition of elementary siphons that contains both 16 and 19, let  $U' = U \setminus \{19\}$ .  $U'$  has the same interface as  $U$  (since 16 and 19 have the same interface) and it is worse than  $U$  as it is composed of less ingredients. Otherwise, let  $U'$  be the set of duals to the ones contained in  $U$ . 19 is not contained in  $U'$  as we assumed that  $16 \notin U$ .  $U'$  has the same interface as  $U$  as dual elementary siphons have the same interface.  $U'$  includes traps for the same interfaces as traps are symmetric w.r.t. exchanging  $c$  and  $d$ . After having removed redundant siphons, the remaining information for  $N_1 \oplus N_2$  is

- $\Sigma = \{13, 14, 16, 17, 22, 23\}$ ;
- $int(13) = \{a\}, int(14) = \{b\}, int(16) = \{a, e\}, int(17) = \{a, f\}, int(22) = \{b, e\}, int(23) = \{b, f\}$ ;

- $L_Q = \{\{a, e\}, \{a, f\}, \{b, e\}, \{b, f\}, \{e\}, \{f\}\};$
- $L_{M1} = \emptyset;$
- $w_Q(\{a, e\}) = \{\{16\}, \{17, 22\}\}, w_Q(\{a, f\}) = \{\{17\}, \{16, 23\}\}, w_Q(\{b, e\}) = \{\{22\}, \{16, 23\}\}, w_Q(\{b, f\}) = \{\{23\}, \{17, 22\}\}, w_Q(\{e\}) = \{\{16\}, \{22\}\}, w_Q(\{f\}) = \{\{17\}, \{23\}\};$
- $w_M = \emptyset.$

This means that the number of elementary siphons as well as the number of interfaces to be considered for traps does not increase during composition. There are only minor differences in  $w_Q$ .  $\square$

#### 4.4 Discussion

The approach projects information about a component to its interface. The calculations at the interface, e.g. finding matchings or redundancies, appear to be complex, but their complexity depends much more on the size of the interface than on the size of the net behind the interface. In the running example, we may compose longer and longer chains or rings of components such as in Fig. 11. As each resulting component has information similar to the one for single components, the overall complexity grows linearly with the number of components to be composed. In comparison, the resulting net has an exponentially growing number of minimal siphons (since every circle where either the upper or the lower place is taken forms a minimal siphon). We conclude that the divide-and-conquer approach is beneficial at least in those cases where a decomposition exists such that intermediate interfaces during re-composition remain small. How to obtain such a decomposition in general remains to be seen. Since we may switch to the original algorithm for computing elementary siphons and traps at any stage of decomposition, it is possible to apply the divide-and-conquer strategy whenever the size of the interface between components is significantly smaller than the inner structure of a component. Consequently, we may judge that the proposed strategy is rather valuable even though we cannot provide experimental evidence at this time.

## 5 Conclusion

We proposed two new approaches to deciding the siphon trap property and thus to getting information about important properties like liveness or deadlock freedom. One approach is a straight transformation to the SAT problem for which we inherit the sophistication of existing SAT solvers. The second approach uses the well known divide-and-conquer strategy. It is based on a known decomposition into open nets which we refined such that we may arrive at arbitrarily small components, and on a projection of information about siphons and traps to the interface of a component. This way, complexity expresses itself more in terms of the size of interfaces than of the size of the component as such which makes the procedure applicable at least for certain classes of nets with somewhat sparse connectivity.

For the first approach, the main remaining issue is to get to smaller formulae. In particular, we frequently copy a certain subformula. There may be structural considerations for reducing the number of required copies for certain net classes. In the divide-and-conquer approach, there are still some non-deterministic choices. We need to provide a prototype implementation including heuristics for these choices for further underpinning its usefulness. In addition, we would like to have a reasonable criterion for verifying Def. 10.

## References

1. Barkaoui, K., Minoux, M.: A polynomial-time graph algorithm to decide liveness of some basic classes of bounded Petri nets. In: Jensen, K. (ed.) ICATPN 1992. LNCS, vol. 616, pp. 62–75. Springer, Heidelberg (1992)
2. Desel, J., Esparza, J.: Free Choice Petri nets. Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press, Cambridge (1995)
3. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
4. Esparza, J., Nielsen, M.: Decidability issues for Petri nets. Petri Nets Newsletter 52, 245–262 (1994)
5. Hack, M.H.T.: Analysis of Production Schemata by Petri Nets. Master’s thesis, MIT, Dept. Electrical Engineering, Cambridge, Mass (1972)
6. INA. Integrated Net Analyzer (2003), <http://www2.informatik.hu-berlin.de/~starke/ina.html>
7. Karatkevich, A.: Analysis by solving logical equations – calculation of siphons and traps. In: Dynamic Analysis of Petri Net-based Discrete Systems. LNCIS, vol. 356, pp. 87–93. Springer, Heidelberg (2007)
8. Mennicke, S., Oanea, O., Wolf, K.: Decomposition into open nets. In: AWPN 2009. CEUR Workshop Proceedings, vol. 501, pp. 29–34. CEUR-WS.org (2009)
9. MiniSat. Minimalistic, open-source SAT solver (2007), <http://www.minisat.se>
10. Minoux, M., Barkaoui, K.: Polynomial algorithms for proving or disproving Commoner’s property in Petri nets. In: Proceedings 9th Workshop on Theory and Applications of Petri Nets, vol. 1, pp. 113–125 (1988)
11. Zaitsev, D.A.: Decomposition of Petri nets. Cybernetics and Systems Analysis (5), 131–140 (2004)

# AlPiNA: A Symbolic Model Checker<sup>\*</sup>

Didier Buchs, Steve Hostettler,  
Alexis Marechal, and Matteo Risoldi

Software Modeling and Verification laboratory  
University of Geneva  
Route de Drize 7, 1227 Carouge, Switzerland  
<http://smv.unige.ch>

**Abstract.** AlPiNA is a symbolic model checker for High Level Petri nets. It is comprised of two independent modules: a GUI plugin for Eclipse and an underlying model checking engine. AlPiNA's objective is to perform efficient and user-friendly, easy to use model checking of large software systems. This is achieved by separating the model and its properties from the model checking-related concerns: the users can describe and perform checks on a high-level model without having to master low-level techniques. This article describes the features that AlPiNA provides to the user for specifying models and properties to validate, followed by the techniques that it implements for tuning validation performance.

**Keywords:** System design and verification, Higher-level Nets Models, Algebraic Petri Nets, State Space Generation, Computer Tools for Nets, Model Checking.

## 1 Introduction

*Model checking* consists of verifying whether a model satisfies a given property, usually expressed using temporal or modal logic. Model checking implies fully automated property proving. When a property does not hold on a model, the user should get a counterexample. High-level formalisms [1,2] allow users to specify complex models in an easier way. At the same time model checking can benefit from the richness of information of such models. In our approach, we use High-Level Petri nets (HLPNs) – more precisely, a class of HLPNs called *Algebraic Petri nets* (APNs) [2]. In APNs, the model is composed of a Petri net – expressing aspects related to causality, non-determinism and concurrency – and of algebraic abstract data types (AADTs, commonly called ADTs with a slight abuse) – describing the data and their manipulation.

This article explains how to perform model checking on APN using our tool called *AlPiNA*, which stands for **Algebraic Petri Net Analyzer**. AlPiNA's objective is twofold. First, to perform reachability analysis on finite models in a

---

<sup>\*</sup> This work was partially funded by the COMEDIA project of the Hasler foundation, ManCom initiative project number 2107.



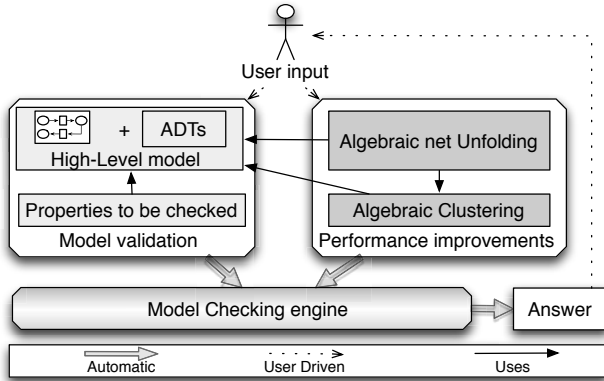


Fig. 1. Overview of the ALPiNA framework

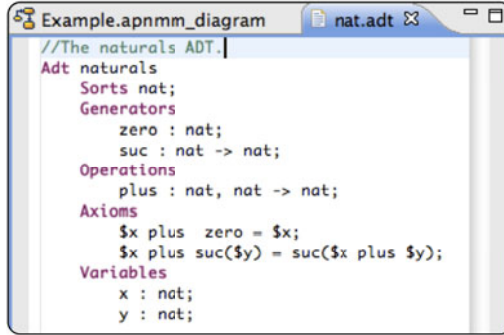
user-friendly and efficient way. Second, to scale up these models without requiring the end user to know the underlying model checking techniques used by ALPiNA, like its decision diagrams-based symbolic representation [3].

The article is organized as follows: section 2 presents a general overview of the framework. Section 3 describes the definition of models and properties. In section 4 we introduce two notions, *algebraic clustering* and *partial net unfolding*, that can be used to improve model checking efficiency. Section 5 describes ALPiNA's architecture, and gives some hints about the tool development process. Section 6 shows the performance of our tool compared to two other well-known model checkers, for some common examples found in the literature. Finally, section 7 draws conclusions and outlines future perspectives.

## 2 Framework Overview

Modeling with ALPiNA requires the user to treat two different concerns, shown in Fig. 1. The first concern is specifying a model along with the properties to be checked (left block). The model specification consists of algebraic data types and a Petri net describing the system behavior. Properties are logical statements concerning the model's states, that can be either satisfied or not. In our case, a property is said to be satisfied by a model if it evaluates to *true* for all of the model's states (and thus is called an invariant). Specifying a model and its properties is already enough to perform checks if the model has a small number of states.

However, for certain problems with high concurrency, the number of states may quickly become too large to be represented as the model size increases if no optimization is performed. Efficiently checking models with larger state spaces requires the user to provide additional information about the model's structure. This constitutes the second user concern: specifying algebraic clustering and algebraic net unfolding to improve model checking performance (right block of



```

//The naturals ADT.
Adt naturals
  Sorts nat;
  Generators
    zero : nat;
    suc : nat -> nat;
  Operations
    plus : nat, nat -> nat;
  Axioms
    $x plus zero = $x;
    $x plus suc($y) = suc($x plus $y);
  Variables
    x : nat;
    y : nat;

```

Fig. 2. Naturals ADT

Fig. 1. Both user-provided information are then merged by the engine to perform a more compact computation of the state space.

### 3 Model Creation and Validation

AIPiNA is fully integrated in the Eclipse environment. Models are created in a dedicated Eclipse project, and some example projects are provided with the tool. The model and the properties to be checked are defined through a mix of graphical and textual editors. This corresponds to the left block of Fig. 1.

#### 3.1 Data Types Definition

APNs can be compared to the well-known colored Petri nets [1], but they replace *colors* with algebras defined using ADTs. Informally, an ADT allows the definition of a set of values using inductive axiomatization. This axiomatization enables some automatic processing on data types, which is very helpful for some features presented here, like algebraic clustering. Every value has a given *sort* and can be described with a *term*, i.e., a combination of the ADT's *operations* and *generators*. The behavior of operations is defined using a set of *axioms*. In AIPiNA, axioms are treated as term-rewriting rules [4] – they are repeatedly applied until a normal form is reached, i.e., no rules can be applied anymore. AIPiNA offers a helper where one can specify a term and ask for its normal form.

Fig. 2 shows one of the most basic ADT defined in AIPiNA: naturals. The naturals ADT defines a sort, called `nat`. There are two *generators* for this sort, `zero` and `suc`, that are combined to represent all the values in the algebra. This definition is similar to the Peano axiomatization. We also define one operation, `plus`, with two axioms that define its behavior. AIPiNA's data types offer more than this example shows, like sub-sorting, polymorphism or data type modularity similar to what is shown in [5].

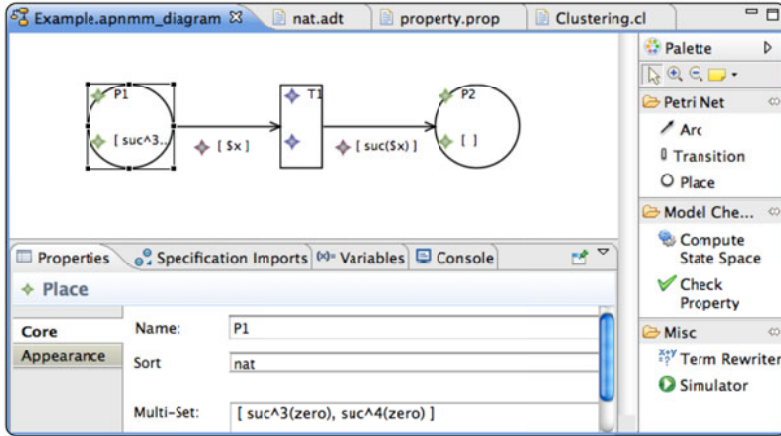


Fig. 3. AIPiNA’s GUI for defining an APN

### 3.2 Control Flow Definition

Once the data types have been defined, one has to model the system’s behavior. *Petri nets* are a widely used formalism to model concurrent systems like communication protocols. In short, a Petri net is made of *Places*, that represent system resources, and *Transitions*, the firing of which represents system state evolution. Places and Transitions are connected with arcs. AIPiNA offers both a graphical and a textual interface to define algebraic Petri nets, where places, transitions and arcs are annotated with algebraic terms. Fig. 3 shows the graphical interface. A tool palette allows the user to create Petri net elements easily, and properties of the net elements can be edited using the standard Eclipse *properties* view.

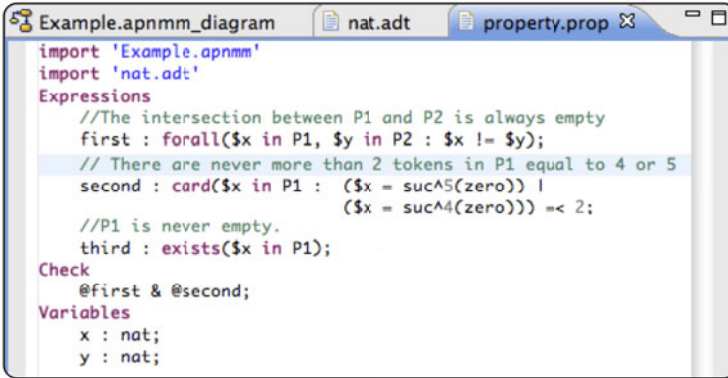
### 3.3 Property Checker

Once a model is complete, one can perform model checking. AIPiNA offers a dedicated textual language for specifying reachability properties, equivalent to first-order logic. This language was inspired by Helena’s [6], a well-known model checker for HLPNs, but was adapted to profit from the flexibility of ADTs.

A property in AIPiNA is a boolean expression that must be evaluated to *true* for every state of the model. If there is a reachable state where the property does not hold, a textual representation of this state is returned as a counterexample. Fig. 4 shows a property definition in the actual AIPiNA editor.

## 4 Performance Improvements

Due to the state space explosion problem, advanced techniques are needed to perform checks on large models. We propose *partial net unfolding* and *algebraic clustering*. These techniques correspond to the right block of Fig. 1.



```

Example.apnmm_diagram  nat.adt  property.prop
import 'Example.apnmm'
import 'nat.adt'
Expressions
//The intersection between P1 and P2 is always empty
first : forall($x in P1, $y in P2 : $x != $y);
// There are never more than 2 tokens in P1 equal to 4 or 5
second : card($x in P1 : ($x = suc^4(zero)) |
                                ($x = suc^5(zero))) <= 2;

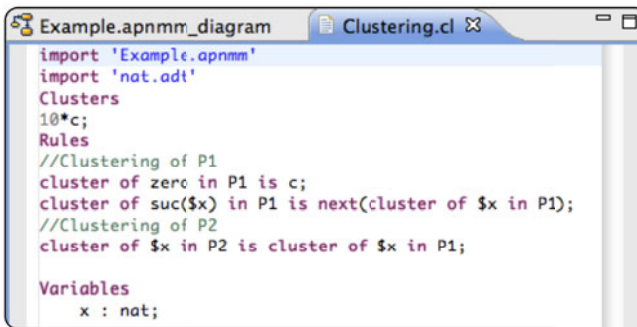
//P1 is never empty.
third : exists($x in P1);
Check
@first & @second;
Variables
x : nat;
y : nat;

```

Fig. 4. Property definition example

#### 4.1 Algebraic Clustering

Concurrency and non-determinism are major causes of the well known state space explosion problem. This happens when model components have no causal dependencies among them, i.e. when they may evolve independently. Algebraic clustering exploits the inductive definition of data types to define these components, improving model checking performance. The goal of clustering is to calculate the state space of individual model components, and then to combine them. To specify clustering, we assign each term and place of the net to a cluster. Existing works already use control flow to perform clustering. We extend that technique to data types.



```

Example.apnmm_diagram  Clustering.cl
import 'Example.apnmm'
import 'nat.adt'
Clusters
10*c;
Rules
//Clustering of P1
cluster of zero in P1 is c;
cluster of suc($x) in P1 is next(cluster of $x in P1);
//Clustering of P2
cluster of $x in P2 is cluster of $x in P1;
Variables
x : nat;

```

Fig. 5. Clustering definition

In order to exploit the inductive nature of values, algebraic clustering can be itself defined using induction. Criteria for choosing clusters are generally guided

by the structure of the model and the properties to be verified. In general, the best results are obtained when independent elements are put in separate clusters. A heuristic is to put processes in the same cluster as their resources, while resources shared among several processes are put in another cluster. By doing this, the operations that only affect one process can be performed within a single cluster. This speeds up the computation.

AIPiNA provides a language for defining clustering. Consider a system with ten processes identified by natural numbers. Each process has two states, modeled by two places P1 and P2. Using the mentioned heuristic, each process is assigned to its own cluster. Fig. 5 shows such a clustering function. Each of the ten first natural numbers is added to its own cluster in both places P1 and P2.

From a user perspective, choosing the granularity of the clustering is a trade-off between the independence of the components and the size of the cluster. A too fine or too coarse clustering will lead to sub-optimal performance.

## 4.2 Algebraic Net Unfolding

Another feature designed to increase the model checking performance in AIPiNA is algebraic net unfolding. Some data types, like naturals or lists, have a sort with an infinite domain; others, like the booleans, are finite. Algebraic unfolding consists of explicitly enumerating ADT values before computing the state space. AIPiNA can use this information to improve the speed of state space generation. From a user perspective, the novel aspect of this feature resides in the fact that for each data type, users can choose whether or not they want the engine to unfold it (partial net unfolding) and if so whether the unfolding should be bounded (bounded sort unfolding). This choice is a trade-off between the complexity reduction of the state space generation and the cost of the unfolding operation itself, which is polynomial with respect to the size of the algebras.

AIPiNA is able, to a certain extent, to guess what a good unfolding combination could be, avoiding combinations that may lead to an incomplete state space coverage. Fig. 6 shows the dialog for specifying unfolding.

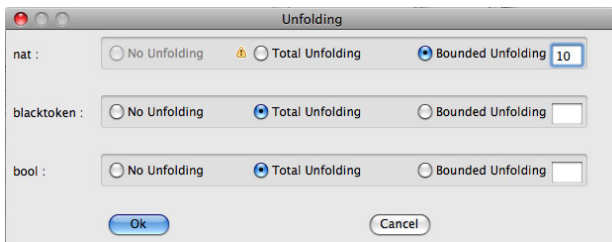


Fig. 6. Algebraic unfolding example

As it can be seen, some choices are grayed-out by the system in order to prevent the user from specifying an incorrect unfolding. This system guess uses the following rules:

- An enumerated data type can be totally unfolded (i.e. the `bool` in our example).
- If a data type appears in the clustering definition, it *must* be unfolded.
- If a data type is infinite, it cannot be totally unfolded. If the system cannot be sure whether an inductive data type is infinite a warning is displayed. Fig. 6 shows this for the naturals `nat`.

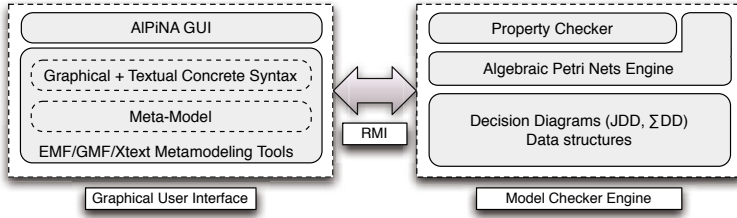
The user must be careful when choosing a bound for unfolding. If the bound is too big, the unfolding may become more expensive than the model checking itself. On the other hand, if the bound is too small, the validation may become incomplete. In this case, the result is inconclusive.

## 5 Architecture

One of ALPiNA's goals is providing a synergy between usability and performance. The EMF framework provides a good platform to create the user interface; a powerful underlying model checking engine allows checking models with large state spaces. To provide a great user experience we leverage the Eclipse platform. It provides a well-known user interface model as well as very efficient tools to develop new software. The most natural way to create a model is using a language that is specifically tailored to the domain of the model – a Domain Specific Language (DSL). The Eclipse platform provides several tools to develop DSLs, one of the most relevant being the Eclipse Modeling Project (EMP) [7], which follows a metamodeling approach. The EMP platform includes the following technologies:

- EMF <http://www.eclipse.org/modeling/emf/>: we used the Eclipse Modeling Framework to define ALPiNA's metamodels. The main advantage of EMF is that the large palette of bundled tools makes the creation and manipulation of metamodels easy, while providing strong integration between them.
- GMF <http://www.eclipse.org/modeling/gmf/>: the Graphical Modeling Framework allows the creation of a graphical editor for EMF metamodels. The ALPiNA graphical editor is entirely based on GMF.
- XText <http://www.eclipse.org/Xtext/>: XText allows the creation and manipulation of textual DSLs, and generates a complete textual editor with features like code completion, syntax highlighting and on-the-fly syntax checking. ALPiNA's textual editors were created using XText.

The real power of these tools is that, being based on the same building blocks, they are well integrated. This allows the graphical editor to reference elements from the textual editors (e.g. the terms can be used in the APN) and vice versa (e.g. the places of the APN can be used in the clustering definition). Moreover,



**Fig. 7.** AIPiNA’s architecture

the metamodeling approach allows the integration with other projects that use the same technology. As an example, we are currently working on the integration of the PNML language [8]. PNML’s goal is to become a standard for defining different types of Petri nets. It can be used as a platform for the interchange between different Petri net tools. Like AIPiNA, PNML was developed using EMF. This allowed us to use metamodel transformations to perform the translation between PNML and AIPiNA’s language.

As a research project, AIPiNA tries to be as modular as possible in order to support the rapid evolution of technologies as well as new ideas.

Fig. 7 describes the layered architecture of AIPiNA. The left block represents the structure of the graphical user interface (GUI). The first layer manages the user interface itself. It leverages the code that has been produced by the tools on the second layer, which presents the metamodeling tools used in AIPiNA: the concrete syntaxes are defined with GMF and XText, based on metamodels created with EMF.

The right block of Fig. 7 represents AIPiNA’s model checking engine, which performs the actual computation. The first two layers are the property checker and the APN engine. These two layers act as an interface to the engine: the input is an APN and some properties to be checked; the output is the result of the property checks and some information about the APN’s state space, such as the computation time and the number of states. These two layers are based on the underlying data structures, called Decision Diagrams [9,10,11], used to calculate and represent the state space.

Communication between the blocks of Fig. 7 is done through Java Remote Method Invocation (RMI). This ensures a strong independence between GUI and engine, and allows experienced users to easily extend the tool: both the interface and the engine can be substituted by different components or re-used in other projects.

## 6 Benchmarks

To validate AIPiNA’s approach, we compared its results with Maria’s [12] and Helena’s [6], two well-known model checkers. Table 1 shows some benchmarks for the state space generation by the three tools. These benchmarks were performed

**Table 1.** State space generation (empty=N/A; --=out of memory failure)

		AlPiNA				Maria		Helena	
		Partial Unfold.		Total Unfold.					
Model Size	States #	Mem (MB)	Time (s)	Mem (MB)	Time (s)	Mem (MB)	Time (s)	Mem (MB)	Time (s)
Distributed Database									
10	197E3	10	0.8	12.4	1.3	47	44.3	24	9
15	7.2E7	33	2.6	41	5.8	-	-	1.4E3	7.5E3
35	5.8E17	544	69.4	789	278	-	-	-	-
Leader Election									
10	31302			10.3	0.72	20	3.4	10	7
15	399E4			27.7	1.4	795	361	107	142
50	1.7E21			702	76	-	-	-	-

on a 2.5GHz Intel Core 2 Duo Macintosh, with a limit of 1GB RAM assigned. All source codes are available at <http://alpina.unige.ch>. Maria and Helena are clearly outperformed when clustering and unfolding are enabled. In the other case (clustering disabled), AlPiNA's results are of the same order of magnitude.

Both examples in Table 1 are well-known communication protocols often used in the literature of the model checking field. The Distributed Database model is very interesting because it proves that partial unfolding can be more efficient if applicable. The Leader Election protocol does not have infinite domains, and therefore partial unfolding does not make sense. We can see in these examples that AlPiNA is able to handle a much bigger state space than the other tools.

The performance gain of AlPiNA is more evident on models that have very strong concurrency. This is the case of the Leader Election protocol in Table 1, where the different processes are more independent than in the Distributed Database protocol. In such models the state space is close to the cartesian product of each model component's state space. In the best case, memory consumption is logarithmic with respect to the number of states.

## 7 Conclusion

This article provided an overview of AlPiNA, a high-level Petri nets based model checker. We showed its user-friendly interface, and gave some details about the tools used to create it. Benchmarks were given to show the state space computation performance. More detailed technical information and benchmarks were published in [13].

The tool is in active development and will be significantly improved within the next months. In the future we plan on adding the following features:

- Automatic test case generation [14] using the generated state space.
- CTL support. Currently, only reachability properties are allowed. Supporting CTL would also allow us to build traces to a given counterexample.



- Object orientation and encapsulation as was done in [5]. A hierarchical view of the model would simplify the modeling activity.
- A DSL for process properties for automatic cluster inference.

ALPiNA, its source code and metamodels are available under the GPL license as an Eclipse plugin or a complete Eclipse package at <http://alpina.unige.ch>.

## References

1. Jensen, K.: Coloured Petri Nets. Springer, Berlin (1996)
2. Reisig, W.: Petri nets and algebraic specifications. *Theoretical Computer Science* 80, 1–34 (1991)
3. 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)
4. Dick, A.J.J., Watson, P.: Order-sorted term rewriting. *Comput. J.* 34(1), 16–19 (1991)
5. Buchs, D., Guelfi, N.: A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering* 26(7), 635–652 (2000)
6. Pajault, C., Evangelista, S.: High level net analyzer, <http://helena.cnam.fr/>
7. Eclipse. Eclipse modeling project, <http://www.eclipse.org/modeling/>
8. Weber, M., Kindler, E.: The Petri Net Markup Language. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) *Petri Net Technology for Communication-Based Systems*. LNCS, vol. 2472, pp. 124–144. Springer, Heidelberg (2003)
9. Couvreur, J.-M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.-A.: Data decision diagrams for Petri net analysis. In: Esparza, J., Lakos, C.A. (eds.) *ICATPN 2002*. LNCS, vol. 2360, pp. 101–120. Springer, Heidelberg (2002)
10. Buchs, D., Hostettler, S.: Sigma Decision Diagrams: Toward efficient rewriting of sets of terms. In: Corradini, A. (ed.) *TERMGRAPH 2009: Preliminary Proceedings of the 5th International Workshop on Computing with Terms and Graphs*, number TR-09-05 in *TERMGRAPH workshops*, pp. 18–32. Università di Pisa (2009), <http://smv.unige.ch/publications/pdfs/termgraph09.pdf>
11. Hostettler, S.: Java decisions diagrams library. Technical Report 201, CUI, Université de Genève (June 2008), <http://smv.unige.ch/technical-reports/pdfs/TR201-JDD.pdf>
12. Mäkelä, M.: Modular reachability analyzer, <http://www.tcs.hut.fi/Software/maria/>
13. Buchs, D., Hostettler, S.: Toward Efficient State Space Generation of Algebraic Petri Nets. Technical Report 206, CUI, Université de Genève (January 2009), <http://smv.unige.ch/technical-reports/pdfs/TR206-APNClustering.pdf>
14. Buchs, D., Lucio, L., Chen, A.: Model checking techniques for test generation from business process models. In: Kordon, F., Kermarrec, Y. (eds.) *Ada-Europe 2009*. LNCS, vol. 5570, pp. 59–74. Springer, Heidelberg (2009)

# Wendy: A Tool to Synthesize Partners for Services

Niels Lohmann<sup>1,2</sup> and Daniela Weinberg<sup>1</sup>

<sup>1</sup> Universität Rostock, Institut für Informatik, 18051 Rostock, Germany

<sup>2</sup> Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

wendy@service-technology.org

**Abstract.** Service-oriented computing proposes *services* as building blocks which can be composed to complex systems. To reason about the correctness of a service, its communication protocol needs to be analyzed. A fundamental correctness criterion for a service is the existence of a *partner service*, formalized in the notion of *controllability*.

In this paper, we introduce *Wendy*, a Petri net-based tool to synthesize partner services. These partners are valuable artifacts to support the design, validation, verification, and adaptation of services. Furthermore, Wendy can calculate an *operating guideline*, a characterization of the set of all partners of a service. Operating guidelines can be used in many application scenarios from service brokerage to test case generation. Case studies show that Wendy efficiently performs on industrial service models.

## 1 Objectives

The emerging field of service-oriented computing (SOC) proposes to build complex systems by composing geographically and logically distributed services. A service encapsulates a certain functionality and offers it through a well-defined interface. As services are not executed in isolation, their communication protocol has to be considered when reasoning about the correctness of a service. To this end, *controllability* [20] has been introduced as a fundamental correctness criterion for services. A service is controllable iff there exists a *partner service* such that their composition is compatible, for instance free of deadlocks. Controllability not only proves correctness of a service, but a partner service also provides insight into the communication behavior of the modeled service. Furthermore, it is a useful artifact to validate [11] or document the service and is the basis of adapter synthesis [3].

A controllable service usually has more than one partner service. An *operating guideline* finitely characterizes the (possibly infinite) set of all partner services of a service [12]. Operating guidelines are useful in a variety of applications including test case generation [4], service correction [6], instance migration [5], or service substitution [17]. They further allow to efficiently realize a service-oriented architecture in which the service provider publishes his operating guideline at a

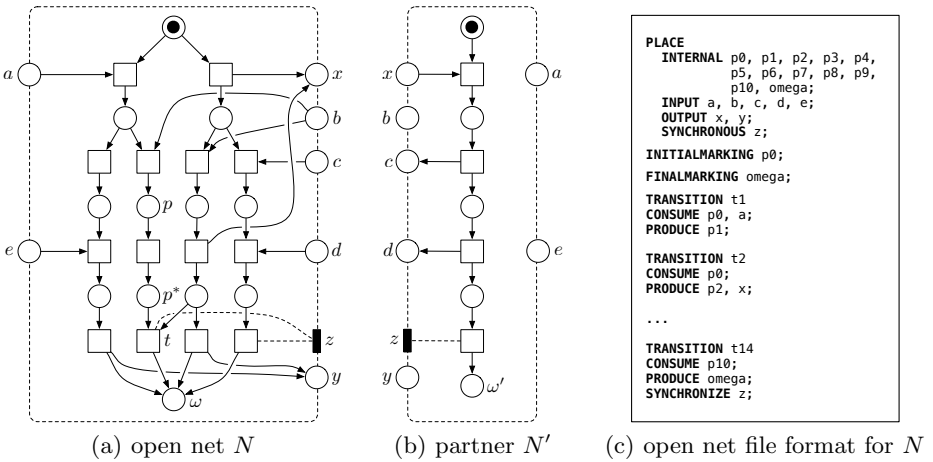


Fig. 1. Open net  $N$  (a), a partner open net  $N'$  of  $N$  (b)

service broker. A service requester then only needs to check whether his service is one of the partner services which is characterized by the operating guideline [10].

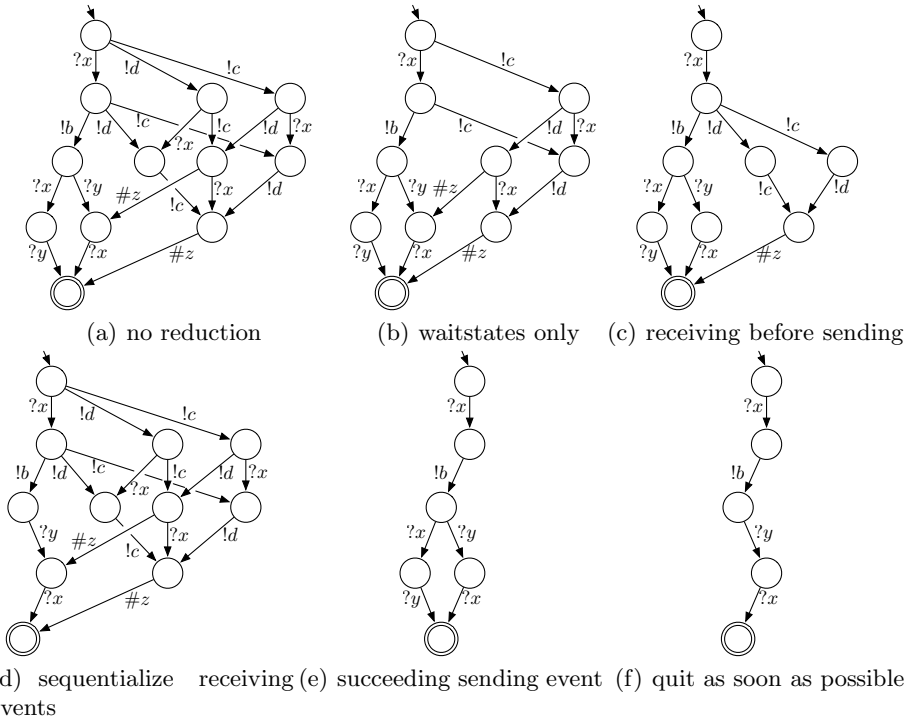
In this paper, we introduce Wendy, a tool to synthesize partner services and to calculate the operating guideline of a service. Wendy provides the basis of a vast variety of applications which are essential in the paradigm of SOC. Case studies show that Wendy can cope with industrial and academic service models. We continue with sketching the functionality and the used formalism. The architecture of Wendy and the components it is built of are described in Sect. 3. Section 4 shows how Wendy can be used in different use cases, provides experimental results, gives information about how to obtain Wendy, and discusses improvements with respect to a previous implementation. Section 5 concludes the paper.

## 2 Functionality

### 2.1 Background

The theory [12,20,18] implemented by Wendy focuses on the behavior (both control flow and communication protocol) of a service. We model a service as an *open net* [13], a special type of Petri net with an interface. Open nets can be automatically derived from industrial service description languages such as WS-BPEL [7].

Figure 1(a) and 1(b) depict two open nets  $N$  and  $N'$ . The interfaces (modeling the message channels of the service) are depicted on the dashed frames. We distinguish asynchronous input and output message channels (modeled as places) and synchronous message channels (depicted as black rectangles). To differentiate desired *final markings* from unwanted deadlocks, an open net has a



**Fig. 2.** Different types of partners of  $N$  by applying a certain reduction rule

distinguished final marking ( $[\omega]$  for  $N$  and  $[\omega']$  for  $N'$ ). We require the interface places to be empty in the initial and final marking.

The open nets  $N$  and  $N'$  can be *composed* by merging the input places of  $N$  with the output places of  $N'$  (and vice versa), and by fusing each pair of transitions of  $N$  and  $N'$  which are connected to the same synchronous channel. Initial and final markings are added element-wise. The composition  $N \oplus N'$  is *compatible*: the only reachable deadlock  $[\omega, \omega']$  is a final marking. If  $N$  and  $N'$  are compatible, we call both  $N$  and  $N'$  *controllable* [20], and refer to  $N$  as a partner service of  $N'$ , and vice versa.

### 2.2 Partner Synthesis

Wendy analyzes controllability of an open net and synthesizes a partner as a witness if the net is controllable. This partner is an automaton model which can be transformed into an open net using known tools such as Petrify [2]. In the automaton representation, asynchronous send actions, asynchronous receive actions, and synchronous actions are preceded by “!”, “?”, and “#”, respectively.

The open net  $N$  is controllable, and Fig. 2(a) shows the partner synthesized by Wendy. By design, this partner is *most-permissive* in the sense that it simulates any other partner including  $N'$ . This partner reveals that no compatible partner

of  $N$  will ever send an  $a$ -message. The transition connected to the input channel  $a$  and the transition connected with channel  $x$  are in conflict. It can never be ensured that  $N$  will always first receive an  $a$ -message whenever it is available. It may as well send an  $x$ -message which leads the net into the right hand branch where no  $a$ -message will ever be received. Consequently, the  $a$ -message remains on the message channel and the final marking becomes unreachable.

The validation of  $N$  can be refined by applying behavioral constraints [11] which filter the set of partners, for instance to only those partners sending a  $b$ -message.

### 2.3 Operating Guidelines

Although every partner of  $N$  is simulated by the most-permissive partner, the converse does not hold. To characterize exactly the set of all partners of  $N$ , we annotate the states of the most-permissive partner with Boolean formulae (see [12] for details). This annotated most-permissive partner is called an *operating guideline*. Wendy can calculate an operating guideline by generating the Boolean formulae in a postprocessing step.

### 2.4 Reduction Rules

Wendy synthesizes different types of partners depending on the analysis goal: (1) by applying no reduction rules, it synthesizes a most-permissive partner which serves as the basis for the calculation of the operating guideline; (2) for checking controllability, the type of the synthesized partner is not of much interest. Here, we focus on a quick answer about the mere existence of a partner; (3) by combining the reduction rules in a certain way, Wendy generates a particular partner which will be used later on, for instance for generating an adapter service [3].

The first three reduction rules (cf. Fig. 2(b)–(d)) focus on reducing the overhead due to asynchronous communication, whereas the last two rules (cf. Fig. 2(e)–(f)) always lead to small partners and hence a quick answer with respect to controllability. See [18] for further information on the reduction rules.

- *Waitstates only (WSO)*. Send a message or synchronize only if it is necessary for  $N$  to move on. Messages are not sent in advance.
- *Receiving before sending (RBS)*. Before sending a message or synchronization, receive every asynchronous message sent by  $N$ .
- *Sequentialize receiving events (SRE)*. Receive the messages sent by  $N$  in a certain order again. This, however, does not necessarily have to be the order the messages have been sent by  $N$ .
- *Succeeding sending event (SSE)*. Quit any interaction as soon as one sent message leads to a proper interaction with  $N$ .
- *Quit as soon as possible (QSP)*. Quit the interaction if any (synchronous or asynchronous) action has led the interaction with  $N$  to a proper end.

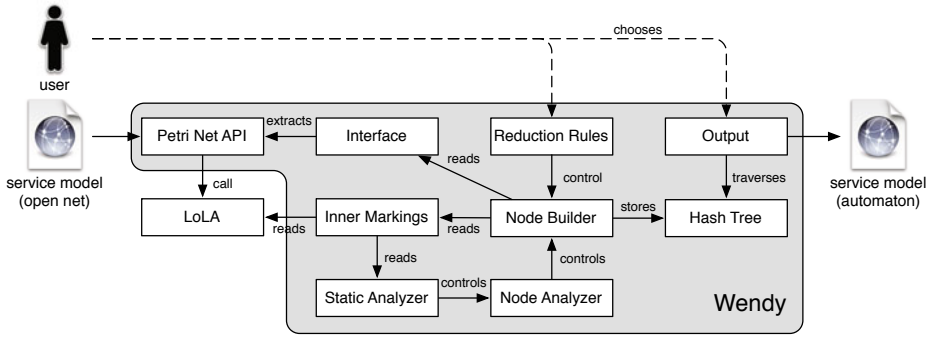


Fig. 3. Architecture overview

The reduction rules may be combined arbitrarily. However, only a few combinations are reasonable, whereas other combinations are less appropriate. In the following, we list a few types of partners which can be synthesized by combining different reduction rules (shown in brackets at the end of each description).

- *Chatty partners* send as much as possible and receive as less as possible (SRE). These partners are best suited for adapter synthesis [3], because they hardly block the overall system, but send messages as early as possible.
- *Good Listeners* only send messages if it is necessary (WSO, RBS).
- *Arrogant partners* let service  $N$  do all the work for proper interaction and only react if it is necessary (WSO, RBS, SRE).
- *No-talker partners (1)* only react to the actions of service  $N$  and quit as soon as one sent message leads to a proper end (WSO, RBS, SSE).
- *No-talker partners (2)* listen, but as soon as one sent message leads to a proper end, they quit (WSO, SSE).
- *Lazy partners* do not like to interact with  $N$  (WSO, RBS, SRE, QSP).

### 3 Architecture

Wendy is written in C++ and built up out of several components. Figure 3 sketches the overall architecture. To process the input open net, given in a file format as sketched in Fig. 1(c), Wendy uses the Petri Net API<sup>1</sup>, a C++ library encapsulating Petri net-related functions. It extracts the interface of the open net and calls LoLA [19] as an external tool to generate the state space of the *inner* of the open net (i.e., the open net without its interface).

These *inner markings* are then statically analyzed to detect deadlocks as early as possible. For instance, open net  $N$  contains the deadlock  $[p^*]$  of the inner, because transition  $t$  is dead. To synthesize a partner, the node builder calculates an overapproximation (see [20] for details), and the node analyzer removes “bad” nodes. With the help of the information of the static analyzer, every node containing a deadlock — as marking  $[p^*]$  in  $N$  — is declared “bad” right away, and

<sup>1</sup> See <http://service-technology.org/pnapi>

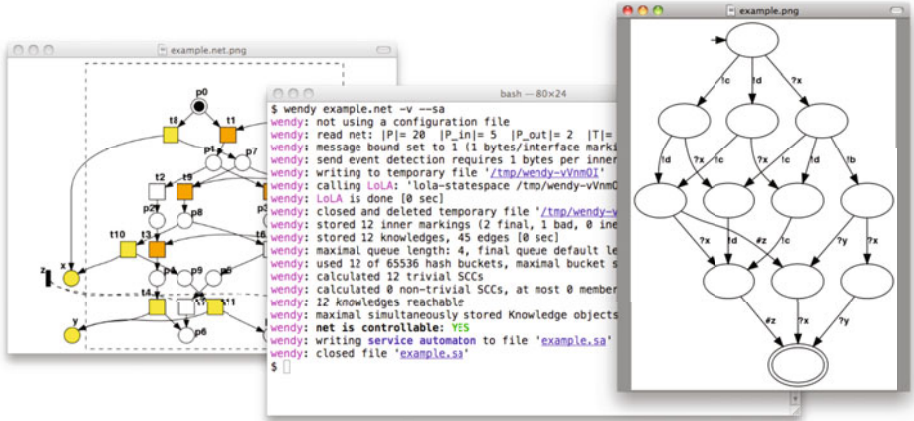


Fig. 4. A screenshot of Wendy analyzing the running example

no more successor nodes of this node are calculated. Inner markings from which a deadlock cannot be avoided anymore (e. g., marking  $[p]$  in  $N$ ) are treated similarly. This *early deadlock detection* has a great impact on the runtime in case an open net contains several deadlocks, for instance due to the application of behavioral constraints [11].

In addition to the node analyzer, the chosen reduction rules influence the synthesis of the partner as well. The calculated nodes are compactly stored in a hash tree to quickly detect already calculated nodes. Finally, the synthesized partner is returned as either an automaton or an operating guideline.

The core design goals of Wendy are to (1) decrease the runtime by gathering as much information about the service model as possible during preprocessing (e. g., by analyzing the inner markings); (2) decrease the memory consumption needed to synthesize partners by implementing very problem-specific abstract data types and by keeping as little information as possible in memory.

## 4 Using Wendy

### 4.1 Use Cases

Wendy is a command-line tool implementing the following use cases. We assume an open net is given as file “service.net” in the format sketched in Fig. 1(C). Alternatively, Wendy can also process PNML files with an extension to model interfaces and final markings.

- *Partner synthesis.* To check if the open net is controllable and to synthesize an unreduced partner if one is present, call Wendy with `wendy service.net --sa`.
- *Operating guidelines.* By invoking Wendy with the following command, the operating guideline of the given open net is calculated: `wendy service.net --og`

- *Reduced partner synthesis.* To synthesize a *no-talker partner* (2) by combining reduction rules *waitstates only* and *succeeding sending event*, call Wendy with `wendy service.net --sa --waitstatesOnly --succeedingSendingEvent`.

In all three use cases, Wendy will print out whether the given open net is controllable or not. If the net is controllable, Wendy generates a file “service.sa” containing the respective partner service automaton or a file “service.og” which contains the operating guideline of the service. With the command line parameter `--dot`, a graphical representation of the output is created (cf. Fig. 4). For uncontrollable nets, the diagnosis algorithm described in [8] is implemented in Wendy. A full description of the command line parameters can be found by executing `wendy --help` or in the manual [2] which also describes the used file formats.

## 4.2 Case Studies

As a proof of concept, we synthesized unreduced and reduced partners of several WS-BPEL services from a consulting company. Each process consists of about 40 WS-BPEL activities and models communication protocols and business processes of different industrial domains. To use Wendy, we first translated the WS-BPEL processes into open nets using the compiler BPEL2oWFn [7].

Table 1 lists details on the processes and the experimental results for the unreduced partner synthesis. We see that the open nets derived from the WS-BPEL processes have up to 15,000 inner markings. The interfaces consist of up to 19 message channels. The number of states of the unreduced partner service (i. e., the most-permissive partner or operating guideline) are sometimes much larger than the original service. The number of transitions grows even faster. The analysis takes up to 300 seconds on a 3 GHz computer with 2 GB of memory. Given that operating guidelines are usually calculated only once and are to be used by the service broker many times, this is satisfactory. It is noticeable that the calculation of the operating guideline’s formulae took only a split of a second for all models.

To further evaluate Wendy, we processed parametrized academic benchmarks within a 2 GB memory limit. Figure 5 illustrates that Wendy is capable of analyzing service models with up to 5,000,000 inner markings and to synthesize partner services with up to 4,000,000 states. At the same time, we see that the largest industrial models we analyzed (14,990 inner markings and 57,996 partner states) do not come close to these bounds.

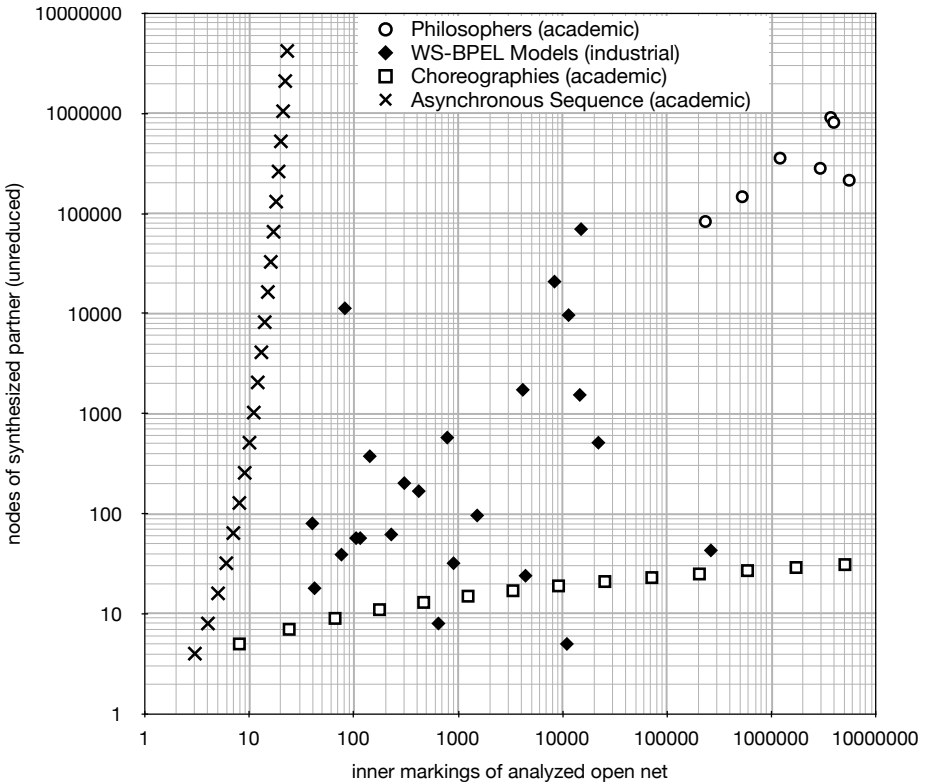
Table 2 summarizes the effect of the reduction rules. Depending on the applied reduction rule, the analysis time can be dramatically reduced. Using the rule *quit as soon as possible*, for instance, allows to calculate a partner service in at most 2 seconds for all services. This result can be generalized to any other open net we analyzed so far: if the inner is bounded (controllability is undecidable if the inner is unbounded [14]), Wendy could always decide controllability by applying reduction rules.

<sup>2</sup> Available at <http://service-technology.org/files/wendy/wendy.pdf>



**Table 1.** Unreduced partner synthesis for industrial WS-BPEL services

service	analyzed service			synthesized partner		
	inner markings	transitions	interface	states	transitions	time
Quotation	602	1,141	19	11,264	145,811	0
Deliver goods	4,148	13,832	14	1,376	13,838	2
SMTP protocol	8,345	34,941	12	20,818	144,940	29
Car analysis	11,381	39,865	15	1,448	13,863	49
Identity card	14,569	71,332	11	1,536	15,115	82
Product order	14,990	50,193	16	57,996	691,414	294



**Fig. 5.** Limits of the synthesis algorithm (2 GB of RAM). Beside some industrial WS-BPEL models ( $\blacklozenge$ ), we processed several parametrized academic benchmarks: *Asynchronous sequence* ( $\times$ ) is a family of services with exponential growth of states of the partner services; *Choreographies* ( $\square$ ) are BPEL4Chor choreographies from [9] with an exponential growth of inner markings; *Philosophers* ( $\circ$ ) is a benchmark set of the WODES workshop, see <http://www.wodes2008.org/pages/benchmark.php>.

**Table 2.** Partner synthesis using reduction rules

service	WSO		RBS		SRE		SSE		QSP	
	states	t	states	t	states	t	states	t	states	t
Quotation	6,244	0	1,667	0	2,287	0	52	0	20	0
Deliver goods	1,328	2	89	0	154	0	65	0	16	0
SMTP protocol	3,270	11	4,872	0	16,837	18	30	0	30	0
Car analysis	1,448	47	108	1	96	11	78	28	38	2
Identity card	1,280	74	261	1	48	2	1,280	74	12	0
Product order	57,762	300	741	1	771	3	1,782	9	101	0

### 4.3 Comparison with Other Tools

Tools from classical controller synthesis [16] are hardly comparable to Wendy, because (1) they consider only synchronous communication (whereas Wendy also supports asynchronous communication which is crucial in SOC), (2) make different assumptions on the observability of internal states and events, and (3) do not support a concept such as an operating guideline to characterize sets of compatible partners.

Both the partner synthesis algorithm and the algorithm to calculate an operating guideline for a service have been previously implemented in the tool Fiona [15]. The design goal of Fiona was the combination of several analysis and synthesis algorithms for service behavior. This is reflected by a flexible architecture which aims at the reusability of data structures and algorithms. Although this design facilitated the quick integration and validation of new algorithms, the growing complexity made optimizations more and more complicated. To overcome these efficiency problems, Wendy is a reimplementations of the synthesis algorithms as a compact single-purpose tool. This reimplementations incorporates the experiments made by analyzing performance bottlenecks through improved data structures and memory management, validation of case studies which gave a deeper understanding of the parameters of the models which affect scalability, and theoretical observations on regularities of synthesized strategies and operating guidelines.

In comparison, Fiona could only analyze three out of the six services presented in Table 1 without taking more than 2 GB of memory. For the other services, the analysis was between 5 and 70 times slower than Wendy. In addition to Fiona, Wendy handles synchronous communication and implements two more reduction rules (*succeeding sending event* and *quit as soon as possible*).

### 4.4 Obtain Wendy

Wendy is free software and is licensed under the GNU AGPL<sup>3</sup>. The source code and precompiled binaries can be downloaded from Wendy's Web site at

<sup>3</sup> GNU Affero Public License Version 3, <http://www.gnu.org/licenses/agpl.html>

<http://service-technology.org/wendy>. We tested several platforms including Windows, Mac OS, Linux, FreeBSD, and Solaris. In addition, an online demo version is accessible at <http://service-technology.org/live/wendy> where the examples of this paper can be replayed in a Web browser.

## 5 Conclusion

The functionality provided by Wendy — the synthesis of partners for services — is a basis of a variety of important applications in the paradigm of SOC. To this end, Wendy is already integrated into tools realizing adapter synthesis [3] and instance migration [5]. Case studies show that Wendy can be used in academic and industrial settings.

In future work, we plan to adjust the synthesis algorithm to exploit the multiple cores that are increasingly present in personal computers. In addition, symbolic representations such as BDDs [1] may further help reducing the memory consumption during the synthesis. At the same time, reduction techniques already implemented in LoLA may be applicable when calculating the inner markings of the given open net.

**Acknowledgments.** The authors thank Stephan Mennicke and Christian Sura for their work on the Petri Net API and Karsten Wolf for sharing experience made with LoLA and valuable discussions on the data structures.

## References

1. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers* C-35(8), 677–691 (1986)
2. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *Trans. Inf. and Syst.* E80-D(3), 315–325 (1997)
3. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany (2008)
4. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: Feuerlicht, G., Lamersdorf, W. (eds.) *ICSOC 2008*. LNCS, vol. 5472, pp. 66–78. Springer, Heidelberg (2009)
5. Liske, N., Lohmann, N., Stahl, C., Wolf, K.: Another approach to service instance migration. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) *ICSOC 2009*. LNCS, vol. 5900, pp. 607–621. Springer, Heidelberg (2009)
6. Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) *BPM 2008*. LNCS, vol. 5240, pp. 132–147. Springer, Heidelberg (2008)
7. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) *WS-FM 2007*. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)

8. Lohmann, N.: Why does my service have no partners? In: Bruni, R., Wolf, K. (eds.) *Web Services and Formal Methods*. LNCS, vol. 5387, pp. 191–206. Springer, Heidelberg (2009)
9. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and participant synthesis. In: Dumas, M., Heckel, R. (eds.) *WS-FM 2007*. LNCS, vol. 4937, pp. 46–60. Springer, Heidelberg (2008)
10. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. *Data Knowl. Eng.* 64(1), 38–54 (2008)
11. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) *BPM 2007*. LNCS, vol. 4714, pp. 271–287. Springer, Heidelberg (2007)
12. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: Kleijn, J., Yakovlev, A. (eds.) *ICATPN 2007*. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)
13. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics* 1(3), 35–43 (2005)
14. Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Undecidability of partner existence for open nets. *Inf. Process. Lett.* 108(6), 374–378 (2008)
15. Massuthe, P., Weinberg, D.: FIONA: A tool to analyze interacting open nets. In: *AWPN 2008*. CEUR Workshop Proceedings, vol. 380, pp. 99–104. CEUR-WS.org (2008)
16. Ramadge, P., Wonham, W.: The control of discrete-event systems. *Proceedings of the IEEE* 77(1), 81–98 (1989)
17. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding substitutability of services with operating guidelines. In: Jensen, K., van der Aalst, W.M.P. (eds.) *Transactions on Petri Nets*. LNCS, vol. 5460, pp. 172–191. Springer, Heidelberg (2009)
18. Weinberg, D.: Efficient controllability analysis of open nets. In: Bruni, R., Wolf, K. (eds.) *Web Services and Formal Methods*. LNCS, vol. 5387, pp. 224–239. Springer, Heidelberg (2009)
19. Wolf, K.: Generating Petri net state spaces (Invited lecture). In: Kleijn, J., Yakovlev, A. (eds.) *ICATPN 2007*. LNCS, vol. 4546, pp. 29–42. Springer, Heidelberg (2007)
20. Wolf, K.: Does my service have partners? In: Jensen, K., van der Aalst, W.M.P. (eds.) *Transactions on Petri Nets*. LNCS, vol. 5460, pp. 152–171. Springer, Heidelberg (2009)

# GreatSPN Enhanced with Decision Diagram Data Structures

Junaid Babar<sup>1</sup>, Marco Beccuti<sup>2</sup>, Susanna Donatelli<sup>2</sup>, and Andrew Miner<sup>1</sup>

<sup>1</sup> Department of Computer Science  
Iowa State University

{junaid,asminer}@iastate.edu

<sup>2</sup> Dipartimento di Informatica  
Università di Torino

{beccuti,susi}@di.unito.it

**Abstract.** Decision diagrams (DDs) have made their way into Petri net (PN) tools either in the form of new tools (usually designed from scratch to use DDs) or as enhancements to existing tools. This paper describes how an existing and established tool (GreatSPN) has been enhanced through the use of DDs provided by an existing open-source library (Meddly). We benchmark the enhanced tool and discuss lessons learned while integrating DDs into GreatSPN.

**Category:** Tool paper.

## 1 Introduction

Generalized Stochastic Petri Nets (GSPNs) [1] and Stochastic Well-formed Nets (SWNs) [4] are well-known extensions of Petri Nets (PNs). GSPNs are useful in modeling stochastic delays where transitions are either *immediate* or *timed*, i.e., they fire with a zero or an exponentially distributed delay. SWNs add token identities to GSPNs and the possibility of automatically exploiting symmetries for efficient state space generation.

GreatSPN is a suite of tools for the design and analysis (qualitative and quantitative) of GSPNs and SWNs. First released by the University of Torino in the late 1980's, GreatSPN has been a widely used tool in the research community, and remains so as it provides a breadth of solvers for computing net structural properties, the reachable states (RS), the reachability graph (RG) with and without symmetry exploitation, and performance evaluation indices using either simulation or analytical solution for steady-state and transient measures. While these solvers are efficient in enabling and firing operations and the underlying data structures are optimized, they do not take advantage of *symbolic* (implicit) storage techniques based on decision diagrams (DDs). A question then arises: can the use of symbolic storage techniques improve GreatSPN's performance while retaining *all* of its features? And if so, by how much and at what cost? Our final goal is to have a state-of-the-art tool that supports advanced data structures for solving GSPN and SWN while saving memory and preserving or improving

time. To limit costs and to ensure adequate implementation quality, we decided to use an existing DD library that automatically handles the complex aspects of using DDs such as caching and garbage collection. We selected Meddly [13], a new DD library, which provides a simple interface (in addition to an expert-level interface for low-level access). Our choice was motivated mainly by the variety of types of DDs that Meddly supports, and by certain features of the library (such as the ability to expand the set of possible values for a variable).

The contribution of this work is along two lines: to improve GreatSPN and to show that, with limited effort and care, an existing, structurally complex tool, can be enhanced with DDs through the use of a library. Being the first use of Meddly inside an existing tool, this work can also be seen as a template for integrating Meddly into existing tools.

There are a number of PN tools that use DDs as preferential data structures. To limit the scope of the related work we shall only consider PN tools and in particular (G)SPN tools since their solutions pose some additional challenges to DDs (as discussed later). The interested reader may refer to [14] for an overview of the different variations of DDs employed by different tools.

SMART [18] was the first SPN tool to use DDs for RS generation and, later, for CTMC storage. It uses an efficient technique (*saturation* [5]) for state-space generation, and can store CTMCs in a variety of compact representations (including matrix diagrams and Kronecker algebra [14]). SMART has a number of additional features, like a rich language for model definition (that extends beyond classical PNs), a simulator, and a model checker for non-stochastic temporal logics; however, it lacks a GUI for net definition.

IDD-CSL [10,17] is a tool targeted towards SPNs for system biology. It supports a rich language for transition rate definition but does not seem to allow immediate transitions or inhibitor arcs. It supports model checking of the stochastic logic CSL, and the computation of steady-state or transient performance indices. It belongs to a suite of tools that provide a GUI for defining a PN, and a tool for model checking non-stochastic logic and for computing a variety of structural properties of PNs. IDD-CSL uses IDD (interval DDs) augmented with state indices (Labeled IDD) needed for computing performance indices. [17] discusses the necessary changes made to IDD for CTMC solution.

The data structures and associated algorithms of tools like SMART and IDD-CSL are very efficient in time and space; however, the DDs are embedded in the tools, and are not available to the community through a public library. Also, considering that these tools were designed with DDs in mind, they are usually, and unsurprisingly, faster than the enhanced GreatSPN.

A relevant example of a tool that has been enhanced through the use of DDs is Moebius [15], a tool for a superclass of GSPNs called Stochastic Activity Networks. Moebius allows for easy integration of different solution methods (and formalisms). A DD-based state space and CTMC generator and solver have been added, but according to the manual, the DD solution saves space but has a very high time penalty.

LibDDD [7] is a publicly available SDD [9] library that we considered as an alternative to Meddly. SDD is thought to be an effective data structure for SWNs, and is well suited for cases in which there is insufficient knowledge of the variables (number and domain) that define the state space to be generated. We chose to use Meddly instead since SDDs are more powerful than necessary for GSPNs and as stated by libDDD’s authors, this power comes at a price. Also, libDDD only provides a low-level interface while Meddly provides (in addition to a low-level interface) a simple non-expert interface (which is all we needed).

The rest of the paper is organized as follows: Sec. 2 recalls the analysis engines of GreatSPN. Sec. 3 gives a brief overview of DDs and Meddly. Sec. 4 describes how GreatSPN has been modified to use DDs using Meddly and discusses the memory and time performance. Finally, Sec. 5 concludes the work, summarizes the lessons learned, and describes our future plans.

## 2 Overview of GreatSPN

GreatSPN is a suite of tools for the design and analysis of GSPNs and SWNs. Its analysis modules support the qualitative and quantitative analysis of GSPNs and SWNs through operations like computation of structural properties, state space generation and analysis, and analytical computation of performance indices. The first modules we chose to optimize were the state space enumeration algorithms, since they are common to both state space analysis and computation of performance indices. GreatSPN uses different solvers for GSPN and SWN. We decided to start our enhancement work from the GSPN solvers for two main reasons: first, although they are theoretically simpler than the ones for SWN, they are a difficult test for the integration of Meddly into GreatSPN as the data structures are optimized and not always trivial to manipulate and understand; second, this code is more likely to suffer from software obsolescence, so that a rewriting is definitely beneficial.

GreatSPN follows a classical fixed-point algorithm for reachability graph (RG) generation, shown in Fig. 1, where  $\mathcal{S}$  is the set of visited markings (the reachability set RS at the end), while  $\mathcal{U}$  is the set of unexplored markings. For each marking  $m'$  added to  $\mathcal{U}$ , the list of enabled transitions  $\mathcal{T}_{m'}$  is stored along with  $m'$ . The list  $\mathcal{T}_m$  is utilized while constructing  $\mathcal{T}_{m'}$ , an important optimization for nets with a large number of transitions. Note that line 12 can be removed if only the reachability set, and not the reachability graph, is desired. The algorithm used by GreatSPN is actually more involved: if the Petri net contains *immediate* transitions, the *vanishing* markings are eliminated during generation, producing the set of *tangible* reachable markings; these details are omitted to simplify the presentation. GreatSPN keeps on a separate file a list of all tangible markings reachable from a given vanishing marking, so it is unnecessary to recompute them when re-entering a vanishing marking.

Major data structures for the algorithm include  $\mathcal{S}$  and  $\mathcal{U}$ , while crucial operations include addition to  $\mathcal{S}$ , addition and removal of markings for  $\mathcal{U}$ , the test to decide if a marking already belongs to  $\mathcal{S}$ , and the computation of enabled

GenerateRG(marking  $m_0$ , Petri net  $PN$ )

```

1:  $\mathcal{S} \leftarrow \{m_0\};$ 
2:  $\mathcal{U} \leftarrow \{m_0\};$ 
3: while  $\mathcal{U} \neq \emptyset$  do
4:   Remove some  $m$  from  $\mathcal{U}$ ;
5:   Determine set  $\mathcal{T}_m$  of enabled transitions in marking  $m$ ;
6:   for all  $t \in \mathcal{T}_m$  do
7:     Determine marking  $m'$  reached from  $m$  when  $t$  fires;
8:     if  $m' \notin \mathcal{S}$  then
9:        $\mathcal{S} \leftarrow \mathcal{S} \cup \{m'\};$ 
10:       $\mathcal{U} \leftarrow \mathcal{U} \cup \{m'\};$ 
11:     end if
12:     Add edge  $(m, m', t)$  to  $RG$ ;
13:   end for
14: end while
15: return  $\mathcal{S}, RG$ ;

```

•  $m_0$  is the initial marking

**Fig. 1.** Traditional enumeration algorithm to build the reachability graph

transitions. GreatSPN uses a balanced binary tree (BBT) for  $\mathcal{S}$ , and to further reduce memory,  $\mathcal{S}$  only contains indices to position in a file (called .mark) that stores markings. To save disk space and I/O time, the markings are stored in a compact way, using an encoding algorithm that exploits P-invariants. Each entry in the BBT also contains the list of the transitions enabled in that marking. The unexplored markings in  $\mathcal{U}$  are stored as a list built on the nodes of the BBT, so that for each marking in  $\mathcal{U}$  the list of enabled transitions is readily available. Thus, lines 4 and 5 in Fig. 1 can be executed very quickly, but line 10 requires determining the enabled transitions in marking  $m'$ , as explained above.

GreatSPN builds the *tangible* reachability graph (TRG), i.e., the graph after elimination of vanishing markings. During generation, the TRG edges (cf. line 12 in Fig. 1) are written to a file (called .wnrg), together with the associated rate (computed from the timed transition rate, possibly multiplied by the weight of the immediate path followed). The TRG file (.wnrg) and the compacted marking file (.mark) are the only information (together with the net and performance indices definitions) needed by the numerical solution engines.

To simplify these and other engines, each reachable marking is assigned a unique integer index in the set  $\{0, 1, \dots, |\mathcal{S}| - 1\}$ . GreatSPN uses “discovery order” (the order in which markings are inserted into  $\mathcal{S}$ ) to index the markings. Indices are required for CTMC construction and solution, and a mapping between each marking and its index is required in many contexts of the solution process, a typical example being the need to compute performance indices from the steady state solution vector of the CTMC. GreatSPN never stores explicitly the mapping between a marking and its index: the  $i^{\text{th}}$  marking is in the  $i^{\text{th}}$  position inside the .mark file, and the edges from the  $i^{\text{th}}$  marking are listed in the  $i^{\text{th}}$  record of the .wnrg file. Note that indices are *not* required for (non-stochastic) model checkers: as we shall see, keeping indices may have a significant price.



### 3 Overview of Meddly

Decision diagrams are directed acyclic graphs used to represent functions on a finite number  $K$  of variables, where each variable  $x_k$  can assume a finite number  $n_k$  of values. Nodes are either *terminal* nodes, which have no outgoing edges, or are *non-terminal* nodes, which are labeled with a variable. Different variable types and ranges, and different rules for managing the graphs lead to various forms of DDs. For instance, binary decision diagrams [3] represent boolean functions on boolean variables, of the form  $f : \{0, 1\}^K \rightarrow \{0, 1\}$ .

Meddly, short for Multi-way and Edge-valued Decision Diagram Library, is an open-source software library [13] that supports several types of DDs (varieties relevant to this work are discussed below). All forms of DDs in Meddly eliminate *duplicate* nodes (two nodes that represent the same function) and require *ordering* of nodes (there is a total ordering  $\succ$  on the function variables). Furthermore, the set of possible values for variable  $x_k$  is assumed to be  $\mathcal{D}_k = \{0, 1, \dots, n_k - 1\}$ . In Meddly, an ordered collection of variables with specified sizes is called a *domain*, which we write as  $\mathcal{D} = \mathcal{D}_K \times \dots \times \mathcal{D}_1$ .

A named collection of nodes of a particular variety of DD, and associated with a common domain, is called a *forest*. Within a given forest, Meddly automatically eliminates duplicate nodes using a unique table [2], imposes other forest-specific reduction rules, and handles memory management of the nodes (storing them compactly, garbage collection, etc.). The following types of forests are relevant to this work and are supported in Meddly.

**MDD:** multi-way DD [11], for functions of the form  $f : \mathcal{D} \rightarrow \{0, 1\}$ .

**MTMDD:** multi-terminal MDD for functions of the form  $f : \mathcal{D} \rightarrow \mathcal{R}$  with either  $\mathcal{R} \subset \mathbb{N}$  or  $\mathcal{R} \subset \mathbb{R}$ . The function “return values” are stored in the forest within the terminal nodes [8].

**EV+MDD:** edge-valued MDD for functions of the form  $f : \mathcal{D} \rightarrow \mathbb{N} \cup \{\infty\}$ . The function “return values” are stored in the forest along the edges and are summed together along paths in the graph [12].

Other forms are already supported or are planned for future releases.

An important feature of any DD software is the ability to create new functions through various operations. In Meddly, several operators are supported whose arguments are functions with a common domain. Additionally, Meddly provides operators to create, evaluate, and destroy functions in a forest. The operations relevant to this work are described below, using simplified versions of the functions (rather than the exact function prototypes) to clarify the presentation.

- `createEdge()` builds a function by explicitly stating a return value for a set of variable assignments. Multiple variable assignments and return values may be specified. Any unspecified assignments are assumed to return a default value (normally 0 but dependent on the forest type). Thus, within a forest  $F$ , a call to  $F.createEdge((a_K, \dots, a_1), a, (b_K, \dots, b_1), b)$  produces a representation of function

$$f(x_K, \dots, x_1) = \begin{cases} a & \text{if } x_K = a_K \wedge \dots \wedge x_1 = a_1 \\ b & \text{if } x_K = b_K \wedge \dots \wedge x_1 = b_1 \\ 0 & \text{otherwise} \end{cases}$$

within forest  $F$ .

- **apply()** builds a function by applying some operator on a set of operands. In particular, for an element-wise binary operator  $\oplus$ , the function

$$f(x_K, \dots, x_1) = g(x_K, \dots, x_1) \oplus h(x_K, \dots, x_1)$$

can be obtained by calling **apply**( $g, \oplus, h, f$ ). This assumes that functions  $f$ ,  $g$ , and  $h$  have the same domain (they can be in different forests) and that the operator  $\oplus$  is defined for the range of  $g$  and  $h$ . Similarly, for an element-wise unary operator  $\ominus$ , the function

$$f(x_K, \dots, x_1) = \ominus g(x_K, \dots, x_1)$$

can be obtained by calling **apply**( $\ominus, g, f$ ). Meddly also provides symbolic state space generation algorithms (including a traditional iteration [16] and saturation [5]) by calling **apply** with an appropriate operator.

- **evaluate()** determines, for the representation of function  $f$ , the value of  $f$  for a given set of variable assignments. Specifically, **evaluate**( $f, (v_K, \dots, v_1)$ ) gives the value of  $f(v_K, \dots, v_1)$ .

Meddly automatically uses and maintains a *computed table* to reduce (often significantly) the computational cost of the **apply()** operations [2]. Meddly also allows for variables sizes to be increased as needed; this allows the user to start building a DD without knowing the *final* variables sizes. Finally, Meddly provides an expert-level interface so that users can define their own operations or access advanced features of the library.

## 4 Enhancing GreatSPN

We have experimented with a few different methods, described below, that we have developed (in an incremental manner) to evaluate the effectiveness of the DDs at various stages. For each method, the goal is to replace the existing representation of  $\mathcal{S}$ , namely the BBT and/or the .mark file, with DDs.

### 4.1 Changing Only the RS

As a first step, we utilize the existing explicit state space generation algorithm in GreatSPN, rather than discarding it in favor of an entirely “symbolic” algorithm (such as [5,16]). We assign an ordering to the set of places  $\mathcal{P}$  (based on the order in which places are defined in the input file), and build the function

$$f(x_{|\mathcal{P}|}, \dots, x_1) = 1 \quad \text{iff} \quad \exists m \in \mathcal{S} : m(p_{|\mathcal{P}|}) = x_{|\mathcal{P}|} \wedge \dots \wedge m(p_1) = x_1$$

**Table 1.** Time and memory (Kb) required for generation

N	S	Original			MDD for S		MDD & N		MTMDD		EV+MDD	
		BBT	File	T.	Mem.	T.	Mem.	T.	Mem.	T.	Mem.	T.
Dining philosophers Petri net (PHIL)												
7	$2.4 \times 10^4$	1,175	1,027	8s	51	8s	36	0.13s	3,449	9s	74	18s
8	$1.0 \times 10^5$	4,977	4,976	45s	75	50s	49	0.15s	14,607	38s	111	87s
9	$4.3 \times 10^5$	21,082	23,717	345s	103	411s	64	0.18s	61,872	210s	160	421s
10	$1.8 \times 10^6$	89,304	104,654	31m	139	28m	80	0.24s	262,092	18m	222	34m
11	$7.8 \times 10^6$	—	—	—	185	82m	99	0.35s	1,084,212	78m	299	156m
12	$3.3 \times 10^7$	—	—	—	235	7h	120	0.43s	—	—	392	8h
30	$6.4 \times 10^{18}$	—	—	—	—	—	—	829	10s	—	—	—
40	$1.1 \times 10^{25}$	—	—	—	—	—	—	1,576	32s	—	—	—
50	$2.2 \times 10^{31}$	—	—	—	—	—	—	2,364	1m	—	—	—
Flexible manufacturing system Petri net (FMS)												
4	$1.3 \times 10^5$	6,627	3,037	11s	76	14s	363	6s	14,740	11s	2,397	22s
5	$6.5 \times 10^5$	31,466	14,421	134s	135	63s	775	27s	66,758	81s	9,744	212s
6	$2.5 \times 10^6$	120,940	55,430	7m	218	3m	1,470	1m	246,234	5m	33,068	42m
7	$8.2 \times 10^6$	—	—	—	353	12m	2,536	5m	625,221	19m	97,389	6h
8	$2.3 \times 10^7$	—	—	—	515	32m	4,070	13m	—	—	—	—
9	$6.1 \times 10^7$	—	—	—	—	—	6,179	31m	—	—	—	—
10	$1.4 \times 10^8$	—	—	—	—	—	8,997	1h	—	—	—	—
11	$3.3 \times 10^8$	—	—	—	—	—	12,688	2h	—	—	—	—

as an MDD. The MDD representation for  $S$  is built exactly as described in the algorithm of Fig. 1 if `evaluate( $f$ ,  $m'$ )` is equal to 0, then  $m' \notin S$ ; the operation  $S \leftarrow S \cup \{m'\}$  can be performed by building a function  $g$  with a call to `createEdge( $m'$ , 1)`, where  $g$  represents the set  $\{m'\}$ , and then calling `apply( $f$ , +,  $g$ ,  $f$ )` to union the sets.

To simplify changes in the implementation, we have kept the list structure for  $U$ , which is now a list of pointers to the position in the .mark file. No indices are stored for the markings and therefore it is not possible to generate the reachability graph and the CTMC. However, this method can be used for reachability analysis, and more importantly, it allows us to check the efficiency of the MDD representation with minimal changes to the implementation.

Results on two benchmark Petri nets are reported in Table 1. In the  $N$  dining philosophers (taken from [16]), the numbers of places, transitions, and MDD variables increase linearly with  $N$ , while in the flexible manufacturing system [6], the model parameter  $N$  specifies the number of parts (initial tokens in certain places), and the number of possible values for the MDD variables increases linearly with  $N$ . Experiments were run on an 2.4 GHz AMD Athlon 64-bit processor with 4 GB memory capacity. In the table the ‘‘Original’’ columns refer to the original GreatSPN implementation, and show the memory required in Kilobytes for BBT and for the .mark file, while ‘‘MDD for  $S$ ’’ refers to this first enhancement (use of an MDD for  $S$  and a list for  $U$ ). For the methods based on DDs, the reported memory is the ‘‘peak’’ memory use; the ‘‘final’’ memory use can be less (often substantially so). From the table it is clear that a first

objective has been achieved, since memory consumption is significantly reduced, while time is a bit better than with plain GreatSPN, but still it does not allow us to obtain the huge state spaces that DDs can often achieve. For instance, we cannot solve FMS with  $N = 9$  due to the huge size of the .mark file.

A clear cause for the time bottleneck is that states are added to  $\mathcal{S}$  one at a time, so that execution time is at least linear in the number of states. To overcome this limit, a “symbolic” firing has been implemented, based on a MDD representation of the next-state function  $\mathcal{N}$ , that allows for the efficient determination of all states reachable from the states in the current  $\mathcal{S}$  set in a single firing. In particular, this requires to encode  $\mathcal{N}$  with an MDD that has twice as many variables as the MDD that encodes the RS, since it represents transitions between states. In our implementation we use the inhibition, pre- and post- incidence matrices (available in GreatSPN) to derive  $\mathcal{N}$  for the model. The RS is then generated by Meddly by calling `apply(rsgen, m0,  $\mathcal{N}$ ,  $\mathcal{S}$ )`, which invokes the traditional symbolic algorithm [16] on initial marking  $m_0$  with next-state function  $\mathcal{N}$ , and stores the result in  $\mathcal{S}$ .

Results are reported in columns “MDD &  $\mathcal{N}$ ” of Table II. Note the large saving in time and space for the PHIL model, while for FMS the results are less impressive, which in a way is not surprising: FMS includes priorities for transitions, leading to a more involved implementation of the symbolic firing, which also requires the use of some additional, intermediate, MDD. These intermediate MDDs are included in the peak memory usage reported in the table.

## 4.2 MTMDD or EV+MDD for Reachability Graph

If we wish to build the reachability graph, and the CTMC, we need a new mechanism for remembering the index of each discovered marking in  $\mathcal{S}$ . This can be done by constructing the function (with slight abuse of notation)

$$f(m) = \begin{cases} 0 & \text{if } m \notin \mathcal{S} \\ \text{index of } m & \text{if } m \in \mathcal{S} \end{cases}$$

where the marking indexes are distinct integers in the range 1 to  $|\mathcal{S}|$ , assigned in “discovery order”. This function can be represented either as a MTMDD or an EV+MDD. In either case, the implementation follows the discussion in Sec. 4.1, except when adding marking  $m'$  to  $\mathcal{S}$ , we call `createEdge(m', index)`. Note that this means that we are back to generating one state at a time.

For both solutions (MTMDD and EV+MDD) we have also modified the implementation of  $\mathcal{U}$  to use an MDD. The operation “remove some  $m$  from  $\mathcal{U}$ ” is implemented through a Meddly function `findFirstElement()`, that efficiently finds a function’s “first” set of variable assignments that return a non-zero value; this marking is then removed from  $\mathcal{U}$  via `apply()` with the subtraction operator.

As a consequence the list of enabled transitions is not stored any longer (to do so would require encoding a different function  $f$  with extra variables for the set of enabled transitions). We therefore must determine the enabled transitions by checking them all when exploring a marking, rather than by simply updating

the known enabled transitions when a new marking is entered (in the algorithm of Fig. 1 line 5 becomes more expensive, while line 10 becomes less expensive).

The performance of the implementation of RS with indices, using MTMDD and EV+MDD, is reported in the last columns of Table 1. The MTMDD turns out to be a worst-case scenario: since each marking has a unique index, there will be no sharing of nodes in the MTMDD. The EV+MDD, instead, can be shown to be never worse than the MTMDD, and for many functions, can be exponentially better. However, manipulating EV+MDDs has a higher cost than MTMDDs, since additional operations are required on the edge values. This classical time-memory tradeoff is clearly seen in Table 1: the MTMDD implementation always requires less time and more memory than the EV+MDD implementation. Interestingly, the MTMDD implementation usually outperforms the original one, even without the faster mechanism to determine the enabled transitions.

## 5 Conclusion

In this paper, we described how an existing and established tool, GreatSPN, was enhanced through the use of DDs as provided by an existing open-source library (Meddly). Several important lessons can be taken away from this exercise. The enhancement described has been implemented using only the non-expert interface of Meddly, which implies a limited learning curve, an implementation that is easier and therefore more likely to be error-free, and the possibility to quickly exploit any future development of Meddly. To provide a feeling of how easy it is to experiment with Meddly, the implementation for MTMDD and for EV+MDD differ only in a single parameter in the forest creation.

Our first attempt using MDDs tells us that improvements can be achieved with minimal changes, but that better improvements can be achieved if the generation algorithm is changed to exploit DD strength (e.g., a symbolic algorithm using  $\mathcal{N}$ ), since any savings gained through the optimal use of DDs may be enough to offset the loss of any optimizations that must be discarded.

We have several plans for further enhancement of GreatSPN with DDs. The issue of index storage for RG and CTMC generation and solution requires further investigation. A possible modification is to store the reachability graph directly in a DD; this would allow the use of MDDs (instead of MTMDDs or EV+MDDs) since indexes would not be required until after generation. However, this would require to either convert the DD representation of the CTMC back into the data structure currently used by GreatSPN, or to completely re-implement all the numerical solution engines to use DDs. Either of these requires advancements in Meddly and substantial implementation in GreatSPN. A different line of work is the extension from GSPN to SWN, this however will require the ability to deal with markings that do not have a fixed, bounded structure, for which SDDs may be a better choice than MDDs. As such, we plan to investigate the use of libDDD [9] for state space generation of SWNs and compare with Meddly.

## Acknowledgment

This work is supported in part by the National Science Foundation under grant CNS-0546041.

## References

1. Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. J. Wiley, Chichester (1995)
2. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: 27th ACM/IEEE Design Automation Conference, pp. 40–45. ACM Press, New York (1990)
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. C-35(8), 677–691 (1986)
4. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic well-formed coloured nets for symmetric modelling applications. IEEE Trans. Comput. 42(11), 1343–1360 (1993)
5. Ciardo, G., Lüttgen, G., Miner, A.S.: Exploiting interleaving semantics in symbolic state-space generation. Formal Methods in System Design 31(1), 63–100 (2007)
6. Ciardo, G., Trivedi, K.S.: A decomposition approach for stochastic reward net models. Perf. Eval. 18, 37–59 (1993)
7. LibDDD webpage, <http://move.lip6.fr/software/DDD>
8. Fujita, M., McGeer, P., Yang, J.Y.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design 10(2-3), 149–169 (1997)
9. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical set decision diagrams and automatic saturation. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 211–230. Springer, Heidelberg (2008)
10. IDD-CSL webpage, <http://www-dssz.informatik.tu-cottbus.de>
11. Kam, T., Villa, T., Brayton, R., Sangiovanni-Vincentelli, A.: Multi-valued decision diagrams: theory and applications. Multiple-Valued Logic 4(1-2), 9–62 (1998)
12. Lai, Y.T., Pedram, M., Vrudhula, S.: Formal verification using edge-valued binary decision diagrams. IEEE Trans. Comput. 45(2), 247–255 (1996)
13. MEDDLY webpage, <http://sourceforge.net/projects/meddly>
14. Miner, A., Parker, D.: Symbolic representations and analysis of large state spaces. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 296–338. Springer, Heidelberg (2004)
15. Moebius webpage, <http://www.mobius.illinois.edu>
16. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri Net Analysis Using Boolean Manipulation. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994)
17. Schwarick, M., Heiner, M.: CSL model checking of biochemical networks with interval decision diagrams. In: Degano, P., Gorrieri, R. (eds.) Computational Methods in Systems Biology. LNCS, vol. 5688, pp. 296–312. Springer, Heidelberg (2009)
18. SMART webpage, <http://www.cs.ucr.edu/~ciardo/SMART>

# PNML Framework: An Extendable Reference Implementation of the Petri Net Markup Language

L.M. Hillah<sup>1</sup>, F. Kordon<sup>1</sup>, L. Petrucci<sup>2</sup>, and N. Trèves<sup>3</sup>

<sup>1</sup> Université P. & M. Curie - Paris 6, CNRS UMR 7606 - LIP6/MoVe  
4, place Jussieu, F-75252 Paris CEDEX 05, France

Fabrice.Kordon@lip6.fr, Lom-Messan.Hillah@lip6.fr

<sup>2</sup> LIPN, CNRS UMR 7030, Université Paris XIII  
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France

Laure.Petrucci@lipn.univ-paris13.fr

<sup>3</sup> Cedric, CNAM, 292, rue St Martin, F-75141 Paris Cedex 03, France

nicolas.treves@cnam.fr

**Abstract.** The International Standard on Petri nets, ISO/IEC 15909, provides a formal semantics and syntax to enable model interchange and industrial dissemination. Part 2 defines a concrete interchange format as an XML-based language: PNML. This language is bound to evolve together with future developments of the standard.

This paper presents PNML Framework, a companion implementation of the standard. It provides developers of Petri net tools with a convenient and fast way to implement support of PNML documents. It abstracts away from any XML explicit manipulation and ensures compliance with the standard by using APIs.

**Keywords:** PNML, Petri nets standardisation, metamodels, MDE.

## 1 Introduction and Goals

The International Standard on Petri nets is divided in three parts. The first one deals with basic definitions of Place/Transition, Symmetric, and high-level nets.

The second part, ISO/IEC 15909-2, defines the interchange format for Petri net models: Petri Net Markup Language [8] (PNML, an XML-based representation). This part of the standard was published on November 11, 2009. It is now ready to be used by tool developers in the Petri Nets community.

Now, the standardisation group starts working on the third part. ISO/IEC 15909-3, aims at defining extensions and variations on the whole family of Petri nets. Extensions are for instance the support of modularity, time or probabilities. Variations consider less important semantic changes such as inhibitor arcs, bounded places etc. This raises the need to support such flexibility in the standard.

This paper presents PNML Framework: an API-based Framework to assist tool developers in achieving conformance with the standard. The motivations for PNML Framework are twofold:

- First, it provides tool developers with a programmer-friendly set of APIs which allows them to easily export/import compliant PNML documents. PNML Framework

has been designed as a companion to the standard; it allows tool designers to manipulate Petri Net concepts instead of XML constructs and frees them from XML programming.

- Second, due to part 3, the standard is deemed to evolve and support different kinds of Petri nets. PNML Framework will provide a middleware software layer to cope with consistency of the required variations at the XML level.

The paper is structured as follows. Section 2 describes the Petri nets types metamodeling framework around which PNML Framework is conceptually built. Then it presents the architecture of PNML Framework and its use of the metamodels. Uses of the tool are presented afterwards. Section 3 ends by showing how Petri net tools can interact with PNML Framework. Section 4 reports a typical application example of model translation from PNML to COQ format. Finally, section 5 discusses how the design principles of the standard implemented by PNML Framework allows for flexibility and ability to evolve, strongly required for compatibility with the upcoming Part 3 of the standard.

## 2 PNML Framework Architecture and Services

As a companion to the standard, PNML Framework must support numerous kinds of Petri nets. So its design is based on a structured set of metamodels issued from the standard and describing components in the family of Petri nets.

This section quickly recalls the metamodels architecture that are detailed in [3]. For lack of space, we do not summarise the metamodels in this paper but [3] is available online. We only focus on the overall architecture of PNML Framework and the way this framework is intended to be used.

**Metamodels for Petri nets.** For the standard to be both robust and maintainable, the interchange format should convey structural information for Petri nets while being respectful of their semantic constraints. Thus, part 1 of the standard defines the semantics of several *Petri Net Types* (i.e. P/T, symmetric and high-level nets), while part 2 provides the associated metamodels.

Another challenge is the support of variations and extensions. To meet these issues, a metamodel-based approach as well as associated model engineering techniques, were chosen since they are tooled up and easily accessible. In addition they provide modular and incremental features to handle variations and extensions of these Petri nets types in an elegant manner, preserving their structural relationships.

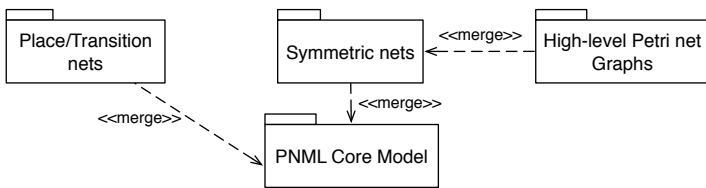


Fig. 1. Overview of the UML packages of PNML



The choice of model engineering techniques is driven by the state of the art of reliable approaches dealing with such issues. Although UML is a semi-formal modelling notation, its flexible levels of abstraction, expressivity, modularity and hierarchy make it appropriate for our goals. This is enforced by the fact that no semantical interpretation of Petri nets is required in an interchange format (only syntax is transferred).

So far, the standard provides a modular and incremental design of Petri net types metamodels. The metamodels of these Petri Nets types are encapsulated in UML packages. Fig. 1 shows their relationships, outlining the incremental design approach.

The *PNML Core Model* package (see Fig. 2) contains the basic structural definition of a Petri net as a labelled directed graph. All type specific information of the net is embedded in the labels. Labels are associated with nodes, arcs or the net. The *PNML Core Model* is intended to be the primary building block upon which concrete Petri net types are defined. Therefore, it imposes no restriction on labels because it is not a concrete Petri net type. For additional details concerning the metamodels, the reader is referred to [3].

As shown in Fig. 1 each concrete Petri net type is built either upon the *PNML Core Model* or upon another existing concrete Petri net type. The *Place/Transition Nets* package thus merges its definitions with the *PNML Core Model* ones, while the *High-level Petri Net Graphs* package merges its own with the *Symmetric Nets* ones. Each concrete Petri net type defines its legal labels by extending the primary definitions in the source package (*PNML Core Model* for *P/T* nets and *Symmetric Nets* for *High-level Petri Net Graphs*) and possibly adding some syntactic restrictions by means of OCL formulae (e.g. connectivity between places and transitions). The metamodels of these labels are

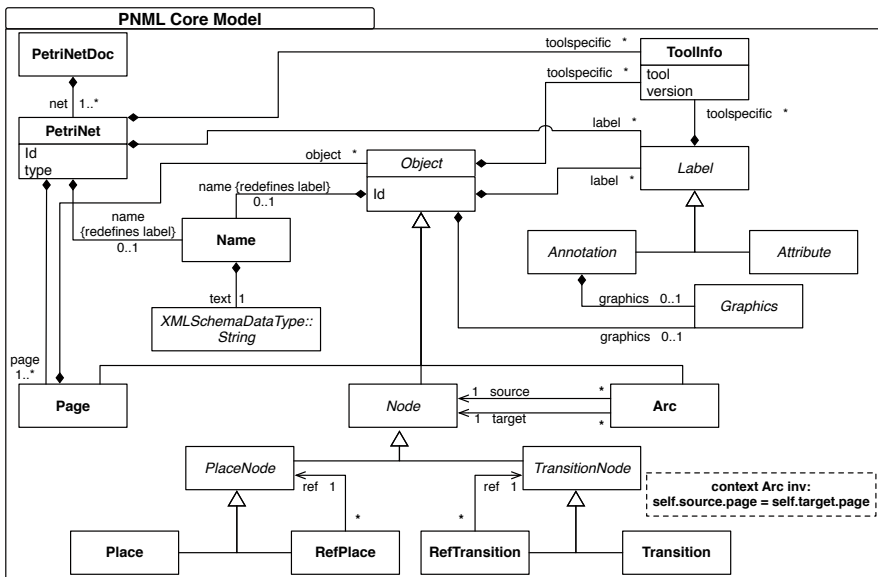


Fig. 2. Overview of the *PNML Core Model* package

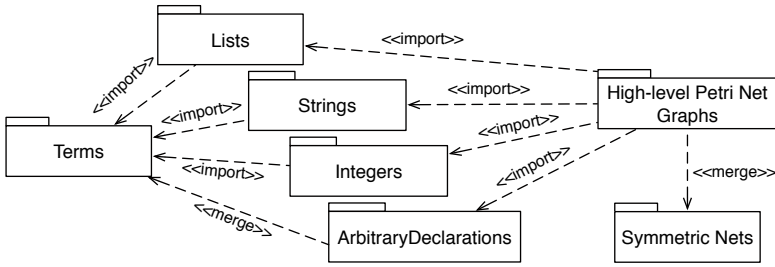


Fig. 3. High-level Petri Net Graphs reusing Symmetric Nets and importing type-specific labels

designed in specific packages so as to be reused as much as possible between related Petri net types.

Fig. 3 illustrates this extension mechanism from Symmetric Nets to High-level Petri Net Graphs. High-level Petri Net Graphs reuse Symmetric Nets definitions but extend their terms by adding new concepts: lists, strings, integers and arbitrary declarations. A more detailed presentation of the standardised Petri net types is provided in [3].

**Architecture of PNML Framework.** Components of PNML Framework are automatically generated from the metamodels in the standard, and thus reuse their structure. We chose to encode these metamodels using EMF [2] in Eclipse. EMF is one of the most advanced and mature model-driven engineering framework, as it supports UML, code generation and model transformation. These features are key characteristics for developing PNML Framework, as well as providing clean modelling facilities and generated APIs (i.e. a set of APIs) for tool developers. Therefore, EMF constitutes a suitable framework to rely on for constructing PNML Framework.

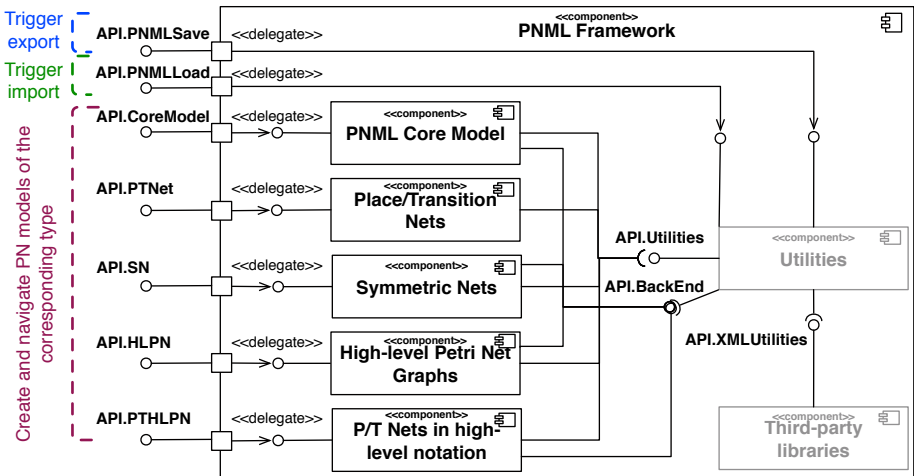


Fig. 4. PNML Framework architecture

Fig. 4 shows the structure of PNML Framework APIs. Each API (left part of the figure) manipulates a given Petri net type and is implemented by a dedicated component. The APIs are named like the corresponding piece of metamodel in the standard. The currently supported Petri net types are: *Place/Transition nets*, *Symmetric nets*, *High-level Petri nets* and *Place/Transition nets in high-level notation*, as defined in part 1 of the standard. Each component provides an API to be used to build models and navigate their structure.

There is no noticeable duplication between the components thanks to the technical UML *merge* operator (shown on Fig. 1) between the standardised Petri nets types in the metamodels. The *merge* operator includes the definitions of an existing Petri nets type into a new one. For instance *High-level Petri Net Graphs* merge *Symmetric Nets* in Fig. 3. As a result, every element previously in *Symmetric Nets* will then be included in the new type.

PNML Framework also embeds *Utilities* that tool developers can use to trigger the loading and storing of models into PNML documents. They can also use this component to turn on or off syntax validation for *PNML Document*. The *Utilities* component is responsible for loading PNML documents and figuring out what type of Petri nets they contain. It also sets up the export of Petri net models into PNML documents and their syntactic validation. This component also provides a workspace (or in-memory repository) where several Petri net models being handled can be stored. *Third-party libraries* are runtime components used by *Utilities*, providing basic APIs to manipulate XML trees. They are shown in grey in Fig. 4.

Most of the PNML Framework code is automatically generated. Manually developed code only concerns the *Utilities* component, which represents 3600 lines of code. This ensures maintainability in order to ease future developments resulting from part 3 of the standard. The next paragraph shows how PNML Framework can be used.

**Uses of PNML Framework.** Creating, for example, a place in any type of Petri net requires a single method call with the associated parameters such as name, marking and position (the method is automatically generated). Then, PNML Framework performs all the appropriate low-level EMF manipulations, in order to minimise the tool developers' efforts. The APIs primitives implement export and import of PNML elements:

- **Export.** Petri net models stored in memory, built as instances of Petri net types (w.r.t. their metamodels defined in the standard) are saved in a file compliant with the PNML syntax. PNML Framework takes care of the process, performing the required checks to produce the PNML document.
- **Import.** Petri net models are loaded in memory, from PNML documents, as instances of Petri net types. PNML compatibility checks are performed when loading models.

Fig. 5 illustrates the export and import mechanisms. It shows a Petri net model on the left-hand side as drawn by a tool user. A typical tool creates the object representation depicted in the centre of the figure. It can be exported by the appropriate API into a PNML file shown on the right-hand side. Importing PNML models is similar.

This process enjoys the independency between the tool internal representation and the current version of PNML. Compatibility concerns are handled by PNML Framework.

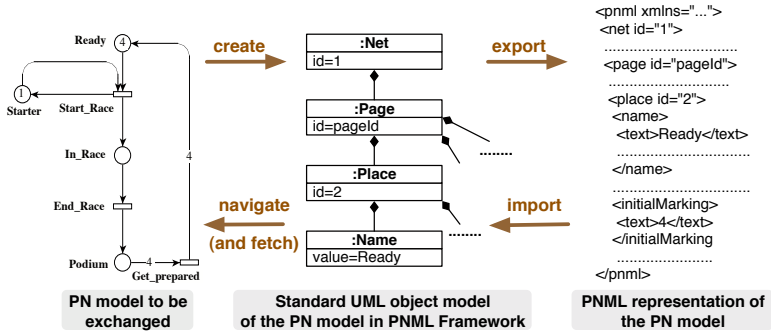


Fig. 5. Export and import of a Petri net model from PNML

**Interaction schemes in using PNML Framework.** PNML Framework is implemented in Java on top of Eclipse. It is also distributed as a standalone library. In order to use PNML Framework, a tool may be implemented in Java (not necessarily on top of Eclipse) or in any language supporting Java bindings. Fig. 6 depicts the ways to use PNML Framework in order to support the interchange standard.

Tool **T1** directly uses the provided API to export/import models in PNML. T1 is the typical example of a standard compliant Petri net tool that relies on PNML Framework. This is the case of Coloane [4].

Tool **T2** exemplifies the use of a standalone application that ensures the conversion of PNML files from/to an existing tool. This converter must parse/produce the T2 internal format. This is also the case of both the CPN-AMI [5] import/export facilities and the PNML2Coq plug-in that is presented in section 3.

In contrast to T1 and T2, tool **T3** relies on its own implementation of the standard. Thus, support of PNML evolutions such as extensions and variations must be handled by T3 developers (with a risk of not conforming to the standard). The PNML web site [8] will be continuously maintained and provide updated versions of PNML.

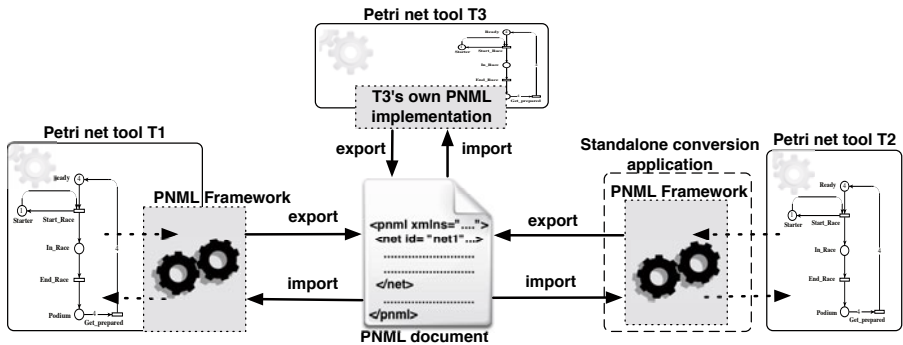


Fig. 6. Petri net tools exchanging PNML documents

```

ModelRepository.getInstance().createDocumentWorkspace("coqWksp"); 1
PnmlImport pim = new PnmlImport(); 2
pim.setFallUse(true); 3
HLAPIRootClass imported = (HLAPIRootClass) pim.importFile("pnmlDocument"); 4
Processor proc = MainProcessor.getProcessor(imported); 5
    {
        if (imported.getClass().equals(  (5)
            fr.lip6.move.pnml.ptnet.hlapi.PetriNetDocHLAPI.class))
        { p = new PTProcessor(); return p;}
    }
proc.process(imported, new PrintWriter(new FileWriter("coqDocument"))); 6

```

Fig. 7. Source code of the PNML2CoQ main function

### 3 Typical Use: PNML to COQ Example

This section illustrates the use of PNML Framework through the export of a Petri net in PNML format into a COQ theorem prover [7] representation as described in [1]. This example is representative of the situation of tool T2 in the previous section, even if it focuses on the import of models from PNML documents only (the export is very similar).

**Overview of the process.** The code snippets of Fig. 7 shows the key instructions to program the import of a *PNML Document*. There are six steps to complete the design of PNML2CoQ (their numbers are shown on the right-hand side of the code):

1. create a workspace in PNML Framework where models can be manipulated (this is an in-memory repository to allow developers to work on several models during the same session),
2. create an *Importer* (from the *Utilities* package) to import a *PNML Document*, this document remains untyped at this stage (*i.e.* it can be any type of Petri net),
3. decide what to do in case the loaded Petri net type is unknown; *e.g.* downgrade to the closest type known by PNML Framework or to a specified one, or raise an error,
4. import the document (no work needed, this is provided by PNML Framework),
5. set the *processor* to be used for the loaded Petri net; this *processor* has to be written by the tool designer (as presented later in this section),
6. the *processor* is called to perform the desired operations (here, generate a COQ file).

**Design of the Processor.** A PNML document can contain several Petri nets, each of them composed of one or more pages. Hence, the *process* code (snippet on the left-hand side of Fig. 8) first handles nets (code on the right-hand side of Fig. 8) and then

```

public void process(HLAPIRootClass rcl, PrintWriter pw){
    PetriNetDocHLAPI root = (PetriNetDocHLAPI) rcl;
    for (PetriNetHLAPI net : root.getNetsHLAPI())
        processNets(net);
}

private void processNets(PetriNetHLAPI ptn) {
    //Some printout into the output Coq file...
    for (PageHLAPI page : ptn.getPagesHLAPI())
        processPages(page);
    //Some printout in the output Coq file...
}

```

Fig. 8. Code snippets from the processor

```
private void processPages(PageHLAPI page) {
    for (PageHLAPI pg : page.getObjects_PageHLAPI())
        processPages(pg);
    for (PlaceHLAPI pl : pth.getObjects_PlaceHLAPI())
        processPlace(pl);
    for (TransitionHLAPI tr : pth.getObjects_TransitionHLAPI())
        processTransition(tr);
    for (ArcHLAPI arc : pth.getObjects_ArcHLAPI())
        processArc(arc);
}
```

**Fig. 9.** Code snippet showing how Pages are handled

```
private void processPlace(PlaceHLAPI pla) {
    StringBuffer sb = new StringBuffer();
    nbplaces++;
    sb.append("Definition " + pla.getId() + " := mk" + "Place" + " " + nbplaces + ".");
    allPlaces = allPlaces + pla.getName().getText() + "::";
    sb.append("\n");
    sb.append("Definition m" + pla.getId() + " := (" + pla.getId() + ",0).");
    initMarking = initMarking + "m" + pla.getName().getText() + "::";
    print(sb.toString());
}
```

**Fig. 10.** Code snippet showing how places are handled

pages (snippet of Fig. 9). The processor uses a writer class to output the resulting COQ syntax into a COQ document (second argument in the *process* signature).

Fig. 9 details the processing of a page. It successively gets enclosed pages, places, transitions and arcs. All processing functions are written by the tool developer, according to his needs. This is eased by the iterators that PNML Framework provides for pages, places, transitions and arcs.

Fig. 10 shows how places are translated into COQ. It handles an object corresponding to a place, *pla*, accessing its attributes through the methods provided by PNML Framework (e.g. *pla.getId()*) so as to construct the output string in the COQ format.

All the examples in the figures above show that models are handled through the provided APIs only. Therefore, the tool developer using PNML Framework does not manipulate any PNML code. The processing we have exposed is fully implemented in PNML2Coq application. It can be reused for another export. In fact, PNML2Coq is inspired from the PNML2Dot tutorial available at [6].

The PNML2Coq application was implemented in one afternoon. The developer had no programming practice with Java but is an experienced programmer.

## 4 Achieving PNML Flexibility and Ability to Evolve

The standard is deemed to evolve (*i.e.* able to support new Petri net types introduced in part 3) and flexible (*i.e.* able to cope with additional information not in the standard). For both, PNML Framework guarantees standard compliance and thus preserves the ability of interchanging models with other tools.

**Mechanisms for Ability to Evolve.** The modular design of metamodels, as well as a compositional and incremental ways to build new Petri net types provide PNML with

the ability to evolve. This capability is crucial for the work on part 3 of the standard — addressing the definition of new Petri net types and structuring constructs.

Since PNML Framework has, from the start, been designed as a companion to the standard, its future enhancements will also follow the advances on the standard. This approach is both valuable for proof of concept purposes as well as future use by tool developers.

The work on addressing the mechanisms for an evolving standard should be fed by the Petri net community long-standing research achievements and recent results. We are therefore actively seeking theoretical as well as practical contributions from practitioners willing to share their new definitions and experiments.

**Mechanisms for Flexibility.** Moreover, flexibility of PNML allows tool developers to cope with tool-specific information in their models which is, of course, not included in the standard. For instance, if a tool associates C code with transitions, it can be introduced as *tool specific information* in the PNML document. PNML Framework supports this provision of the standard.

To do so, PNML Framework provides black box oriented PNML constructs, that allow any non-standard but well-formed XML constructions to be included in a PNML document. These can be included and retrieved by a tool-specific method (the non-PNML XML sentence is encapsulated within tool-specific tags). PNML Framework ensures consistent behaviours in import/export functions.

Thus, to embed some C code in a Petri net for instance, a tool must provide an XML representation of C programs. It might just be the whole C code embedded in a opening and closing XML tag, or a more elaborate syntax tree if the tool developer wants it to have that form.

**Impact on the end-user.** PNML Framework is maintained so as to be standard compliant and also to ensure backward compatibility with its former versions, starting from its current version 2.1, which implements the international standard (2009 version). This is possible thanks to the design choices.

So, if the metamodels evolve, the provided APIs will be regenerated so that the former are backward compatible with the new ones. The management of flexibility is orthogonal and thus not affected by evolution. So, maintenance is not impacted by standard evolution issues. Moreover, PNML Framework is designed to be upward compatible when new upgrades of the standard will appear (*e.g.* when introducing new extensions and/or variations in part 3 of the standard).

If tool developers want to implement their own Petri Net type or extend an existing one, they must provide the framework with its PNML-annotated metamodel, as well as its PNML grammar. Metamodels of the current Petri net types can be used as tutorials. Then, the code handling the new models is automatically generated. We have proceeded in this way to extend the P/T type with inhibitor, reset and read arcs.

## 5 Conclusion

PNML Framework has been designed as a companion and support of ISO/IEC-15909-2, which defines the PNML interchange format. PNML Framework provides a set of APIs

to read and write PNML files. This software is developed thanks to model engineering techniques (here EMF).

PNML Framework has been successfully used to quickly elaborate an export from Petri nets to COQ. The main design steps to build this application demonstrated the simple use of PNML Framework.

PNML Framework is open source and distributed under the Eclipse licence. It is implemented in Java. But as shown in this paper, import/export functions can be quickly developed as a standalone program for tools not being developed in Java.

PNML Framework enjoys flexibility capabilities and ability to evolve, which constitute a major issue in further development of the standard and free tool developers from maintenance issues due to its evolution. Initial successful experiments with small extensions such as inhibitor arcs etc. have assessed these objectives.

**Acknowledgements.** The authors are very grateful to Ekkart Kindler for his support and his comments on earlier versions of this paper.

## References

1. Choppy, C., Mayero, M., Petrucci, L.: Experimenting formal proofs of Petri Nets refinements. In: Proc. Workshop REFINE (associated with FM2008), Turku, Finland, May 2008. Electronic Notes in Theor. Comp. Sci., vol. 214, pp. 231–254. Elsevier Science, Amsterdam (2008)
2. Eclipse Foundation. Eclipse Modeling Framework, <http://www.eclipse.org/emf/>
3. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In: Petri Net Newsletter (originally presented at the 10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools – CPN 2009), October 2009, vol. 76, pp. 9–28 (2009), <http://www.cs.au.dk/CPnets/events/workshop09/assets/paper06.pdf>
4. The Coloane home page (2009), <http://coloane.lip6.fr/>
5. The CPN-AMI home page (2009), <http://www.lip6.fr/cpn-ami>
6. The PNML Framework home page (2009), <http://pnml.lip6.fr/>
7. INRIA. The Coq Proof Assistant home page (2009), <http://coq.inria.fr/>
8. ISO/IEC/SC7/WG19. The Petri Net Markup Language home page (2009), <http://www.pnml.org>



# Author Index

- Babar, Junaid 308  
Beccuti, Marco 308  
Best, Eike 246  
Buchs, Didier 287
- Carmona, Josep 226  
Chatain, Thomas 165  
Choppy, Christine 145
- Darondeau, Philippe 246  
Dedova, Anna 145  
de Frutos-Escrig, David 185  
Donatelli, Susanna 308
- Esparza, Javier 206  
Evangelista, Sami 145
- Fabre, Eric 165
- Gusikhin, Oleg 125
- Hansen, Henri 43  
Harel, David 18  
Hillah, L.M. 318  
Hong, Silien 145  
Hostettler, Steve 287
- Jiao, Li 84  
Juhás, Gabriel 1  
Juhásová, Ana 1
- Kazlov, Igor 1  
Klai, Kais 145  
Klampf, Erica 125
- Kleijn, Jetty 19  
Kordon, F. 318  
Koutny, Maciej 19  
Kristensen, Lars M. 39
- Lê, Dai Tri Man 104  
Leucker, Martin 206  
Lohmann, Niels 297
- Marechal, Alexis 287  
Mending, Jan 63  
Miner, Andrew 308
- Oanea, Olivia 267
- Petrucci, L. 145, 318  
Polyvyanyy, Artem 63
- Risoldi, Matteo 287  
Rosa-Velardo, Fernando 185
- Schlund, Maximilian 206  
Solé, Marc 226
- Trèves, N. 318
- Valmari, Antti 43
- Wang, Yunhe 84  
Weidlich, Matthias 63  
Weinberg, Daniela 297  
Weske, Mathias 63  
Wimmel, Harro 267  
Wolf, Karsten 267