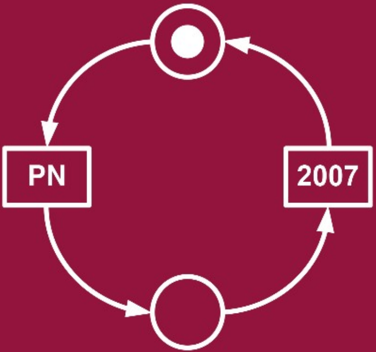
Jetty Kleijn
Alex Yakovlev (Eds.)

LNCS 4546

Petri Nets and Other Models of Concurrency – ICATPN 2007

28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007
Siedlce, Poland, June 2007, Proceedings



Springer

Jetty Kleijn   Alex Yakovlev (Eds.)

Petri Nets and Other Models of Concurrency – ICATPN 2007

28th International Conference on Applications and Theory
of Petri Nets and Other Models of Concurrency, ICATPN 2007
Siedlce, Poland, June 25-29, 2007
Proceedings

Springer

Volume Editors

Jetty Kleijn
Leiden University
Leiden Institute of Advanced Computer Science (LIACS)
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: kleijn@liacs.nl

Alex Yakovlev
Newcastle University
School of Electrical, Electronic and Computer Engineering
Newcastle upon Tyne, NE1 7RU, UK
E-mail: Alex.Yakovlev@ncl.ac.uk

Library of Congress Control Number: 2007928856

CR Subject Classification (1998): F.1-3, C.1-2, G.2.2, D.2, D.4, J.4

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

ISSN 0302-9743
ISBN-10 3-540-73093-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-73093-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12077830 06/3180 5 4 3 2 1 0

Preface

This volume consists of the proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007). 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. In addition, the conferences always welcome a range of invited talks that survey related domains, as well as satellite events such as tutorials and workshops. The 2007 conference had five invited speakers, two advanced tutorials, and five workshops. Detailed information about ICATPN 2007 and the related events can be found at http://atpn2007.ap.siedlce.pl/.

The ICATPN 2007 conference was organized by the Institute of Computer Science at the University of Podlasie and the Institute of Computer Science of the Polish Academy of Sciences. It took place in Siedlce, Poland, during June 25–29, 2007. We would like to express our deep thanks to the Organizing Committee, chaired by Wojciech Penczek, for the time and effort invested in the conference, and to Andrzej Barczak, for all his help with local organization. We are also grateful to the Offices of the Siedlce County Governor and Siedlce City Mayor for their suppport of the local organization.

This year we received 70 submissions from authors from 29 different countries. We thank all the authors who submitted papers. Each paper was reviewed by at least four referees. The Program Committee meeting took place in Leiden, The Netherlands, and was attended by 20 Program Committee members. At the meeting, 25 papers were selected, classified as: theory papers (17 accepted), application papers (5 accepted), and tool papers (3 accepted). We wish to thank the Program Committee members and other reviewers for their careful and timely evaluation of the submissions before the meeting. Special thanks are due to Martin Karusseit, University of Dortmund, for his friendly attitude and technical support with the Online Conference Service. Finally, we wish to express our gratitude to the five invited speakers, Samson Abramsky, Sadatoshi Kumagai, Antoni Mazurkiewicz, Andrzej Tarlecki, and Karsten Wolf, for their contribution to this volume. As usual, the Springer LNCS team provided high-quality support in the preparation of this volume.

April 2007

Jetty Kleijn
Alex Yakovlev

Organization

Steering Committee

Wil van der Aalst, The Netherlands
Jonathan Billington, Australia
Jörg Desel, Germany
Susanna Donatelli, Italy
Serge Haddad, France
Kurt Jensen, Denmark (Chair)
Jetty Kleijn, The Netherlands
Maciej Koutny, UK
Sadatoshi Kumagai, Japan
Tadao Murata, USA
Carl Adam Petri (Honorary Member)
Lucia Pomello, Italy
Wolfgang Reisig, Germany
Grzegorz Rozenberg, The Netherlands
Manuel Silva, Spain
Alex Yakovlev, UK

Organizing Committee

Wojciech Penczek (Chair)
Andrzej Barczak
Stefan Skulimowski
Artur Niewiadomski
Wojciech Nabiałek

Tool Demonstration

Wojciech Nabiałek (Chair)

Program Committee

Jonathan Billington, Australia
Didier Buchs, Switzerland
José-Manuel Colom, Spain
Raymond Devillers, Belgium
Susanna Donatelli, Italy
Jorge de Figueiredo, Brazil
Giuliana Franceschinis, Italy
Luis Gomes, Portugal
Boudewijn Haverkort, The Netherlands
Xudong He, USA
Kees van Hee, The Netherlands
Monika Heiner, Germany
Kunihiko Hiraishi, Japan
Claude Jard, France
Gabriel Juhás, Slovak Republic
Peter Kemper, USA
Victor Khomenko, UK
Jetty Kleijn,
The Netherlands (Co-chair)
Lars Kristensen, Denmark
Johan Lilius, Finland
Chuang Lin, China
Robert Lorenz, Germany
Patrice Moreaux, France
Wojciech Penczek, Poland
Laure Petrucci, France
Michele Pinna, Italy
Lucia Pomello, Italy
Laura Recalde, Spain
Toshimitsu Ushio, Japan
Rudiger Valk, Germany
François Vernadat, France
Karsten Wolf, Germany
Alex Yakovlev, UK (Co-chair)

Referees

Samy Abbes
Gonzalo Argote
Unai Arronategui
Eric Badouel
João Paulo Barros
Marco Beccuti
Marek A. Bednarczyk
Eric Benoit
Robin Bergenthum
Giuseppe Berio
Luca Bernardinello
Henrik Bohnenkamp
Andrzej Borzyszkowski
Anne Bouillard
Roberto Bruni
Nadia Busi
Lawrence Cabac
Javier Campos
Thomas Chatain
Ang Chen
Gianfranco Ciardo
Robert Clarisó
Lucia Cloth
Philippe Darondeau
Massimiliano De Pierro
Jörg Desel
Zhijiang Dong
Boudewijn van Dongen
Till Dörges
Michael Duvigneau
Dirk Fahland
Jean Fanchon
Carlo Ferigato
João M. Fernandes
Mamoun Filali
Paul Fleischer
Jana Flochova
Yujian Fu
Guy Gallasch
Pierre Ganty
Fernando Garcia-Vallés
Gilles Geeraerts
Attilio Giordana
Daniele Gunetti
Serge Haddad
Keijo Heljanko
Jarle Hulaas
David Hurzeler
Agata Janowska
Pawel Janowski
Jens Jørgensen
Michael Köhler
Kathrin Kaschner
Kii Katsu
Kais Klai
Nicolas Knaak
Maciej Koutny
Marta Koutny
Matthias Kuntz
Juan Pablo López-Grao
Linas Laibinis
Charles Lakos
Kai Lampka
Kristian Lassen
Fedor Lehocki
Nimrod Lilith
Niels Lohmann
Levi Lucio
Ricardo Machado
Kolja Markwardt
Peter Massuthe
Sebastian Mauser
Agathe Merceron
Toshiyuki Miyamoto
Daniel Moldt
Salmi Nabila
Apostolos Niaouris
Artur Niewiadomski
Olivia Oanea
Edward Ochmanski
Atsushi Ohta
Ribeiro Oscar
Chun Ouyang
Wieslaw Pawłowski
Luis Pedro
Florent Peres
Denis Poitrenaud
Agata Półrola
Heiko Rölke
Sylvain Rampacek
Jean-François Raskin
Ronny Richter
Matteo Risoldi
Nabila Salmi
Mark Schaefer
Carla Seatzu
Alexander Serebrenik
Dalton Serey
Frederic Servais
Natalia Sidorova
Markus Siegle
Christian Stahl
Martin Schwarick
Maciej Szreter
Shigemasa Takai
Satoshi Taoka
Yann Thierry-Mieg
Simon Tjell
Kohkichi Tsuji
Antti Valmari
Lionel Valet
Robert Valette
Laurent Van Begin
Somsak Vanit-Anunchai
Eric Verbeek
Valeria Vittorini
Daniela Weinberg
Lisa Wells
Matthias Wester-Ebbinghaus
Michael Westergaard
Dianxiang Xu
Shingo Yamaguchi
Satoshi Yamane
Huiqun Yu
Cong Yuan

Table of Contents

Petri Nets, Discrete Physics, and Distributed Quantum Computation

Samson Abramsky

Oxford University Computing Laboratory

Abstract. The genius, the success, and the limitation of process calculi is their *linguistic character*. This provides an ingenious way of studying processes, information flow, etc. without quite knowing, independently of the particular linguistic setting, what any of these notions are. One could try to say that they are implicitly defined by the calculus. But then the fact that there are so many calculi, potential and actual, does not leave us on very firm ground.

An important quality of Petri's conception of concurrency is that it *does* seek to determine fundamental concepts: causality, concurrency, process, etc. in a syntax-independent fashion. Another important point, which may originally have seemed merely eccentric, but now looks rather ahead of its time, is the extent to which Petri's thinking was explicitly influenced by physics (see e.g. [7]. As one example, note that K-density comes from one of Carnap's axiomatizations of relativity). To a large extent, and by design, Net Theory can be seen as a kind of *discrete physics*: **lines** are time-like causal flows, **cuts** are space-like regions, **process unfoldings** of a **marked net** are like the solution trajectories of a differential equation.

This acquires new significance today, when the consequences of the idea that "Information is physical" are being explored in the rapidly developing field of quantum informatics. (One feature conspicuously *lacking* in Petri Net theory is an account of the non-local information flows arising from entangled states, which play a key role in quantum informatics. Locality is so plausible to us — and yet, at a fundamental physical level, apparently so wrong!). Meanwhile, there are now some matching developments on the physics side, and a greatly increased interest in discrete models. As one example, the causal sets approach to discrete spacetime of Sorkin et al. [8] is very close in spirit to event structures.

My own recent work with Bob Coecke on a categorical axiomatics for Quantum Mechanics [4,5], adequate for modelling and reasoning about quantum information and computation, is strikingly close in the formal structures used to my earlier work on Interaction Categories [6] — which represented an attempt to find a more intrinsic, syntax-free formulation of concurrency theory; and on Geometry of Interaction [1], which can be seen as capturing a notion of interactive behaviour, in a mathematically rather robust form, which can be used to model the dynamics of logical proof theory and functional computation.

The categorical formulation of Quantum Mechanics admits a striking (and very useful) diagrammatic presentation, which suggests a link to

geometry — and indeed there are solid connections with some of the central ideas relating geometry and physics which have been so prominent in the mathematics of the past 20 years [3].

References

1. Abramsky, S.: Retracing some paths in process algebra. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 1–17. Springer, Heidelberg (1996)
2. Abramsky, S.: What are the fundamental structures of concurrency? We still don't know! In: Proceedings of the Workshop Essays on Algebraic Process Calculi (APC 25). Electronic Notes in Theoretical Computer Science, vol. 162, pp. 37–41 (2006)
3. Abramsky, S.: Temperley-Lieb algebra: from knot theory to logic and computation via quantum mechanics. To appear in Mathematics of Quantum Computation and Quantum Technology. Chen, G., Kauffman, L., Lomonaco, S. (eds.) Taylor and Francis, pp. 523–566 (2007)
4. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, pp. 415–425 (2004) arXiv:quant-ph/0402130.
5. Abramsky, S., Coecke, B.: Abstract physical traces. Theory and Applications of Categories 14, 111–124 (2005)
6. Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and foundations of typed concurrent programming. In: Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School. NATO ASI Series F, pp. 35–113. Springer, Heidelberg (1995)
7. Petri, C.-A.: State-Transition Structures in Physics and in Computation. International Journal of Theoretical Physics 21(12), 979–993 (1982)
8. Sorkin, R.: First Steps in Causal Sets. Available at http://physics.syr.edu/~sorkin/some.papers/

Autonomous Distributed System and Its Realization by Multi Agent Nets

Sadatoshi Kumagai and Toshiyuki Miyamoto

Department of Electrical, Electronic and Information Technologies
Osaka University
Suita, Osaka 565-0871, Japan
kumagai@eei.eng.osaka-u.ac.jp

Abstract. Autonomous Distributed Systems (ADS) concept plays a central role for designing, operating, and maintaining complex systems in these ubiquitously networked society. Design objectives such as optimality, reliability, and efficiency are renovated according to this system paradigm. Petri nets and its related models provide one of the most concrete basis for realizing ADS to cope with their nondeterministic, concurrent, and asynchronous behavioral features. On the other hand, autonomous decisions by distributed units based on their own interests should be coordinated with total system objectives. Multi Agent Nets are Object-Oriented Colored Petri Nets for implementing autonomous intelligent units and collaborating actions among distributed units. Here in this paper, the realization of ADS by Multi Agent Nets are described through several industrial applications and prototyping that shows paramount versatility of the approach to hardware-software distributed systems encountered in wide variety of engineering problems.

1 Introduction

The history of human beings can be seen as a struggle how to organize the society where individuals tend to enjoy freedom as much as possible. The innovation of systems for organizing such human society has been rather trial and error and there could not find any unilateral evolution in the process. It has been observed that the modern society had based its strength and efficiency on centralized governmental systems. However, from the beginning of 70's since the invention of micro processors, we can realize that all aspects of the society have been changed drastically such as globalization on one hand and diversity of individuals the other. The characteristics of post modern industrial society changed accordingly where agility, flexibility, and robustness are key of core competence. We can say these era as the third industrial revolution as Alvin Toffler correctly cited in his book, "The Third Wave", in 1980. The most significant impact of this innovation, especially on the industrial society, is the enforcement of individual operating units equipped with sufficient information acquisition and processing abilities. On the other hand, the organization extends its scale and complexity without bounds through information and communication technologies (ICT) and rigid centralized control of the total system has been more and more difficult or even obstacle. In Japan, reflecting these tendencies, the intensive research project, called “Autonomous Distribution”, funded by Ministry of Education began in late 80’s. The author was one of the participants and took a role in prototyping the communication architecture and formal analysis of Autonomous Distributed Systems (ADS). Theoretical results such as structural characterization and reachability for live and bounded free-choice nets obtained in [1,2] are by-products of this project. Consecutively to this project, the international research project, called “Intelligent Manufacturing Systems”, funded by Ministry of Trade and Commerce of Japan, has begun in 1992. In this project, we advocated Autonomous Distributed Manufacturing Systems (ADMS). ADMS are composed of variety of autonomous production units and these units can collaborate with each other through communication links to fulfill overall system tasks without any centralized control mechanism. The key features of ADS such as agility, flexibility, and robustness are emphasized in ADMS in addition to its scalable maintainability to cope with product variations of short life cycle in smaller quantities. As a modeling framework, we proposed Multi Agent Nets (MAN) which are implemented among a distributed environment for various applications. MAN is a nested Colored Petri Nets embedded in a Object-Oriented programming similar to other 0bject-0riented nets such as in [3,4]. Comparison of these modeling approach were made in [5]. Formal approach for nested Petri nets was originated by Valk and Moldt [6]. Here in this paper, the key concepts of ADS is summarized in Section 2. In Section 3, main features of MAN is explained. In Section 4, the software environment for MAN to be executed in distributed computers is described. In Section 5, we introduce several industrial applications of MAN to show the versatility of the approach to ADS realization. Section 6 is the conclusion.

2 Autonomous Distributed System Paradigm

ADS is a system composed of distributed autonomous agents (units) where each agent is provided with at least following six capabilities required in the sequence of decision and action process.

1. recognition of whole or partial state,
2. intent proposal based on self interest,
3. total evaluation of the expected results of the intention considering the proposals by other agents,
4. action based on the final decision,
5. timely communication with other agents,
6. learning for improving decision and collaboration strategies.

According to target applications, above functions are specified more precisely, but the fundamental feature of ADS is an iteration of step (2) and step (3) to reach the final consensus and decision that leads to the real action taken by individual agent in step (4) at each time point. Comparing with a centralized system, a drawback of ADS exists in this iteration process, and the efficiency of ADS depends on how quickly the consensus can be obtained among the participated agents.

In the evaluation step (3), each agent must take into account the total optimality and not just an individual interest. In some cases, the decision must be made against own interest. In this sense, we can call the ability in step (3) as a collaboration capability. In many applications, one or two iteration is enough to reach the final decision at each time point because of the simplicity of the negotiation mechanism. In other words, we only need collaborative agents with the common “world view or value” for the objectives of the problem. The learning functions in step (6) speed up the decision process avoiding negotiating iteration steps. Notwithstanding the above drawback, ADS has the obvious advantages such as (1) Flexibility, (2) Agility, and (3) Robustness. Flexibility includes the maintainability and scalability both in design and operation of the system. Frequent changes of the specifications or requirements of the system do not affect the basic architecture of ADS but only require the modification of the function, or simply the number of agent. We do not need to redesign the total system but just need to redesign the function of agent, or just to increase or decrease the number of agents according to the scale change. Behavior of ADS is inherently asynchronous and concurrent. Each agent can act based on its own decision, so agility is a key feature of ADS whereas the time controlling centralized systems cannot realize such autonomous behavior. Like as the scalability of ADS, plug-in and plug-out of agents are flexible so that a damage of a part of agents does not necessarily result in the total system down. The performability is maintained more easily by the remaining agents’ efforts. It implies the Robustness of ADS whereas the centralized counterpart is usually quite vulnerable.

3 Multi Agent Nets

A multi agent net model was shown as a description model for ADS’s in [7]. The objective of the model is to design a system, to simulate on the model, to analyze properties of the system, e.g., performance or dead-lock freeness, on the model, and to control the system by the nets. The model can be used from a design phase through the real time control with application oriented modifications. In ADS’s, each system module has its own controller to decide own behavior. In the multi agent net model, a system is realized by a set of agent nets, and each agent net represents one component of the system. We call the component an agent, and its net representation as an agent net. Behavior of each agent is represented by the net structure of an agent net. Each agent net is an extended colored Petri net (CPN). Figure 1 is an example of the multi agent net. The figure shows a simple communication protocol. A sender sends a message and a receiver replies an acknowledge. When there are one sender and one receiver, the system is represented by a multi agent net with three agent nets. The agent net with CLASS protocol on their left shoulder is the system, and manage the number of senders and receivers and their interactions.

In the multi agent net, interaction between agents is activated by a communication among agent nets, i.e., a rendezvous of agent nets. The agent net with CLASS sender and receiver describe the behavior of sender and receiver, respectively. Agent nets

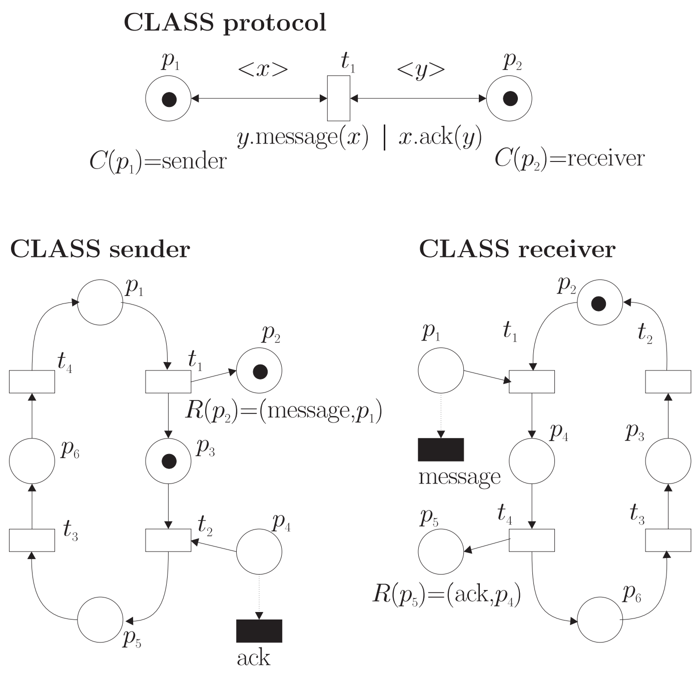


Fig. 1. A Multi Agent Net

belonging to the same agent class have the same net structure, and an agent net is called an instance of the agent class. Each instance has its own net and state, and instances are differentiated by their own ID number.

In a multi agent system, it is undesirable that a change in an agent needs changes in other agents to keep consistency among agents. In order to avoid the propagation, the multi agent net should be based on the object-oriented methodology like [3,4]. Thus, each agent net has two parts: an interface part and an implementation part, and other agent nets are allowed to access only the interface part [7]. This access restriction technique is well known as encapsulation. In the interface part, methods and their input places and output ports are specified on the agent net. Such a place is called an input port (of the method). The implementation part is a net structure without any methods and arcs to methods. A method represented by a bold bar is a fictitious node only expressing an action provided for the requirement of other agent nets. To execute the action, the agent must get tokens in the input places of the method. In Figure 1, a method is represented by a black-filled box, and arcs from input places are drown by dashed arrows. For example, the net "sender" provides a method ack, and the net "receiver" send an acknowledge message via this method. Note that the implementation part is encapsulated from outside. That is, designers of other agent nets can only see that sender provides a method ack and its input place is p4. By using this encapsulation technique, any side effects of changing agent net structure can be reduced or neglected. Note that ordering relations with respect to method call should carefully be examined, since disregards of ordering of method call may cause the deadlock of the system when an agent net has plural methods. In the multi agent net model, each agent class has its own color, and it is allowed that an agent net resides in a place of another agent net as a token [7]. This extension allows us a hierarchical structures in Petri nets. We call it a recursive structure. In Figure 1, the agent net "sender" has its color "sender", and the color specified to place p1 in the agent net "protocol" is also "sender" specified by C(p1)=sender. That is the token in the place p1 in the net "protocol" is the net "sender". Actually, the token has a reference to the net. In this case, the net "protocol" is called an upper-level net, and "sender" is called a lower-level net. Note that there exists an agent net for each token, where the map need not be a bijection. Thus we can consider that the net structure of the lower-level net is one of attributes of the upper-level net, and its marking is the value of the attribute. One of the advantages of this extension is that treating a net as a token enables us to change the number of agents easily. That is, by changing the number of tokens in the agent net "protocol", we can change the number of senders and receivers arbitrarily. A multi agent net can provide reuses of design resources by inheritances and aggregations of classes [7]. When class B inherits class A, class A is called a super-class of class B, and class B is a sub-class of class A. An inheritance of a class implies the inheritance of its color, namely we can say that class B is class A. Therefore, when the classes have colors A and B respectively, a token is specified in a place p1 with C(p1)=A, whether the token implies a net in class A or class B. The reverse, however, is not true. That is, when C(p2)=B, a token that implies a net in class B can exist in p2, but a token that implies a net in class A can not exist. An inheritance and an aggregation of a class are done by copying nets of the super-class. Designers can change a part of the net when the part is copied from the super-class, so long as methods and ordering relations with respect to method calls are kept unchanged. Besides of these extensions including object-oriented methodology, we need to provide intelligence on agent nets by putting a program code on a place or a transition in addition to guards or arc expressions. Such a program code is called an action on a place or a transition. An action on a place is executed when a token is put into the place, and an action on a transition is executed when the transition fires. In order to make program more flexible, agent nets can hold attribute variables. For example the ID number of each agent net, which we mentioned in the previous paragraph, is one of such variables. We can use these variables anywhere in an arc expression, a guard on a transition or in an action. Intelligent decision of each agent can be expressed in these program codes. For the use of real time applications, two kinds of notions of time are introduced into the multi agent net model: a transition invalid time and a token invalid time. Each value is given by an integer number. The transition invalid time inhibits transition firing. A transition with this invalid time cannot fire during that period, once it fires. The token invalid time inhibits use of tokens. A transition can put this invalid time to its output token when it fires. The token cannot be used during that period. We assume that there is unique clock in the system, each transition fires according to the clock. When any invalid time is not given, a transition can fire with zero delay and it can fire again in the same clock if it is enabled, and the succeeding transition can fire with the token in the same clock.

4 Software Environment for the Multi Agent Nets

A software environment for the multi agent nets is shown in Fig. 2 [8]. It consists of three tools and class libraries as follows:

Net Editor: GUI for editing agent nets
Analyzer: It analyzes consistency between agent nets.
Composer: It composes a simulation system of agent nets. Its output is initial state of agent nets.
Class Libraries: Java class libraries for simulation

Java class libraries are improved considering following points:

- actions,
- invalid times, and
- distributed simulation.

The multi agent net must run on distributed computers which are connected on a network. Fig. 3 shows structure of the Class Libraries. In the figure, the dark box is Java class libraries. They are given by vendors. The gray boxes are libraries for the multi agent nets. The lower library is called Multi Agent Environment (MAE) library. It provides a multi agents platform and communication ability of agent nets on distributed computers. In this level, creating and destroying agents, distributing agents to hosts, naming service of agents and communications between agents are supported. The MAE consists of three major components: Agent, AgentBase, and Base-Master. Agent provides a form of agents. The Timer in Fig. 3 is an extension of Agent, and it counts clocks and distributes clocks to agents. Agent-Base provides server service for managing agents. There must be a unique base on each host. When an agent is created, it must be registered in the base on the same host. Base-Master provides server service for naming service of agents and managing bases. There must be a unique master on each simulation. It provides the same service of CORBA name server.

Currently communication between computers is realized by using Java RMI (Remote Method Invocation) on Agent-Base and Agent-Master levels. The higher library is called Multi Agent Net library. It provides component libraries for agent nets, e.g., transitions, places, arcs, tokens, and so on. Agents on the MAE can be agent nets by using these libraries. When some special work on transitions, places or arcs are required, we can put programs to them as “guards”, “actions” or “arc expressions”. The guard and the arc expression must be a boolean function. The action is executed when the transition fires or a token was putted into the place. In the program, we can use attributes of the net and attributes of the relating token. No other attributes can be used, and the program is written in Java. By calling native method in the action by JNI (Java Native Interface), we can control a real system. There may be a case where we want to use general agents in the multi agent system for which a net structure is not necessary to be defined. For example, if an agent only calculates a complex function, no agent nets need to correspond to the agent. These agents may be regular Java agents

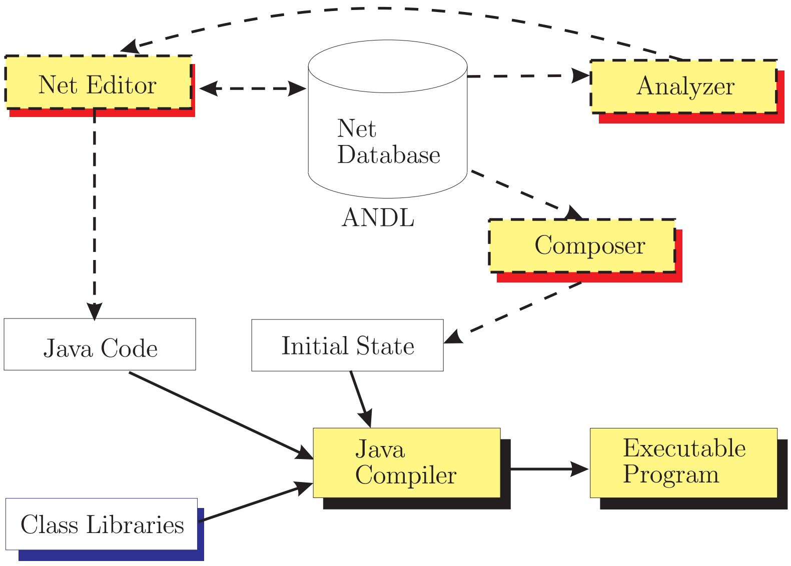


Fig. 2. Software Environment

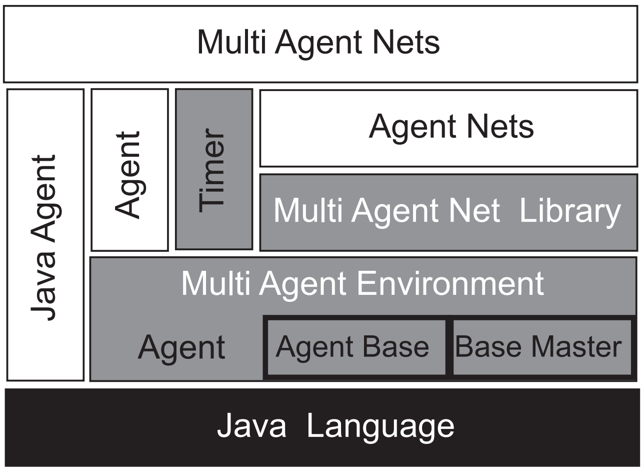


Fig. 3. Class Libraries

or agents on MAE. The Timer is a kind of these agents. To simulate a system, we must prepare Java files for initial states. According to the initial states, agent nets are bound. By starting clock, the simulation starts. Both MAE and MAN libraries have monitoring functions. Thus while the simulation, we can see instantaneous change of status in the system.

5 Realization of ADS by Multi Agent Nets

Once the framework of ADS mechanism is determined and modeled by MAN, we can realize and implement wide varieties of intelligent distributed systems by modifying and specifying according to applications' requirements. Among those applications, we show typical examples in different fields in this section.

5.1 Next Generation Manufacturing [9]

Flexible Manufacturing Systems (FMS) based on ADS concept are designed by multi agent nets. Suppose the FMS consists of machine cells, automated guided vehicles (AGV), information blackboard, and an warehouse. A cell can get an work in constant delay, after it sends a request for AGV to send the work. Each cell can get the following information from the blackboard at any time:

– state of buffers on the warehouse and cells,
– current and previous work on cells,
– state of waiting work,
and
– functional ability of cells.

Taking these information into account, each cell decide the following alternatives autonomously:

– how many and what work to take from the warehouse or from other cells,
– what work to process after the current process is finished,
and
– whether to put works in its own buffer or not.

The warehouse only answer to requests from cells, and decide distribution of works. In order to simplify the problem, we assume that there are enough AGV and routes so that collaboration between AGV is not considered. For designing such system, we need at least three kind of agent nets: the cell, the warehouse and the system. The black board is realized by an Java object in this simulation. Before we design the cell net and the warehouse net concurrently, consensus on communications and cooperation must be established as follows.

– The color of tokens which are passed from a cell to a cell, from a cell to the warehouse, or from the warehouse to a cell is unified.
– Method names in agent nets are also unified.
– Procedure to change stage of the blackboard is unified.
– Delay of transporting a work is set to a constant number.
– The blackboard is an attribute of cells and warehouse. This means the blackboard is a common object of agent nets, and each agent net can refer it at any time. Fig. 4 shows an implementation of a cell. The agent nets consists of six parts:

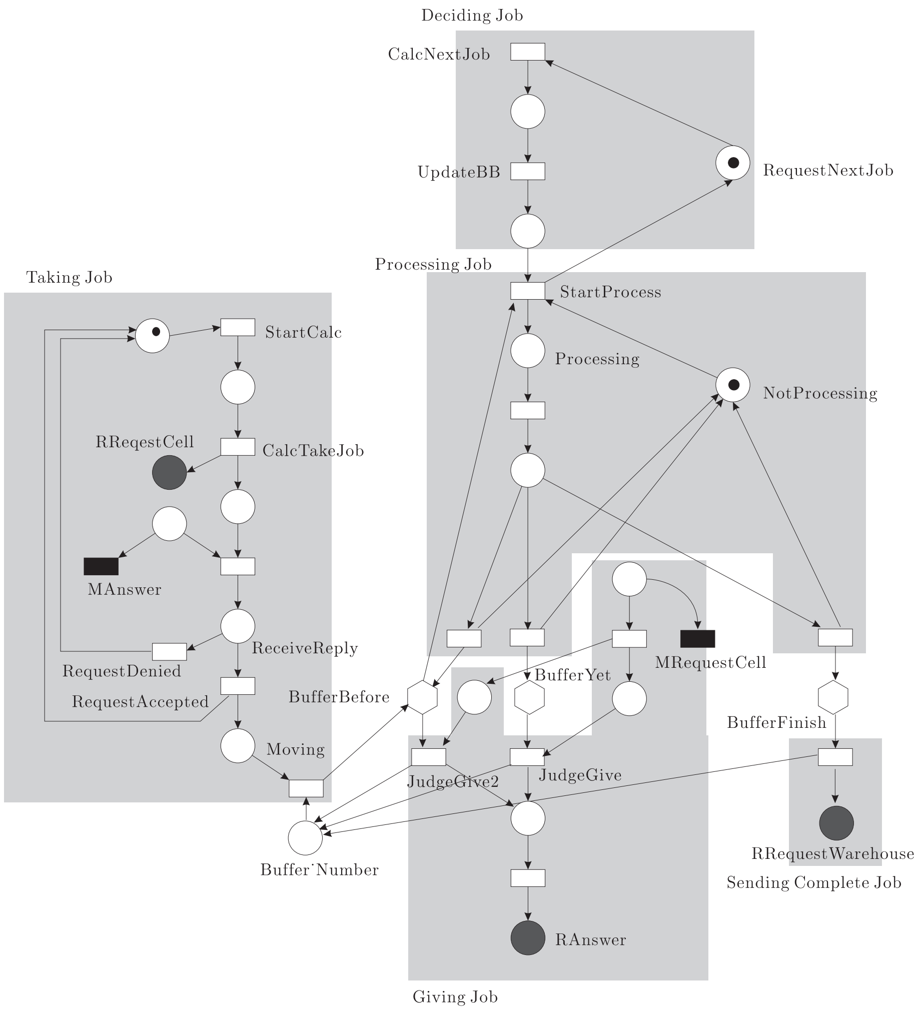


Fig. 4. An Agent Net of a Cell

- Buffers. This part holds works, and consists of four places, BufferBefore, BufferYet, BufferFinish, and BufferNumber.
- Taking jobs part. This part decide jobs to take from other cells or the warehouse (at the transition CalcTakeJob), and send a request. If the request is denied (RequestDenied), the process starts or wait for taking a new work. If the request is accepted (RequestAccepted), the process put new works into the buffer. We put a invalid period to each token in the place Moving, and after the period the token can move to BufferBefore.

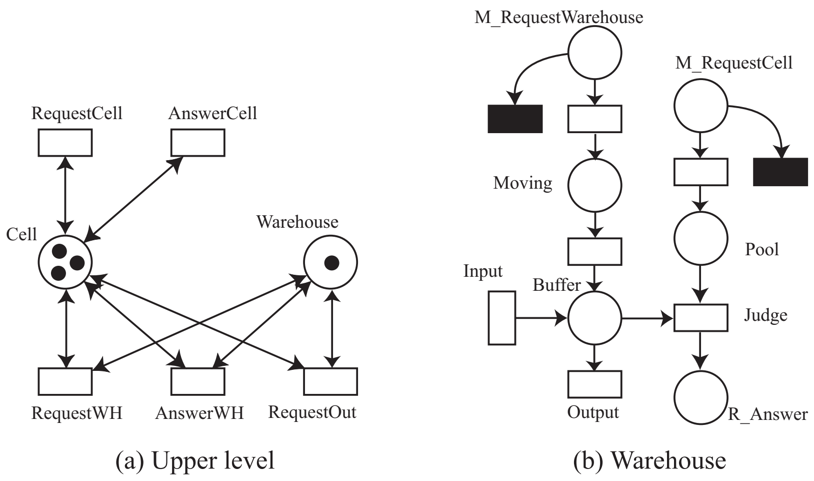
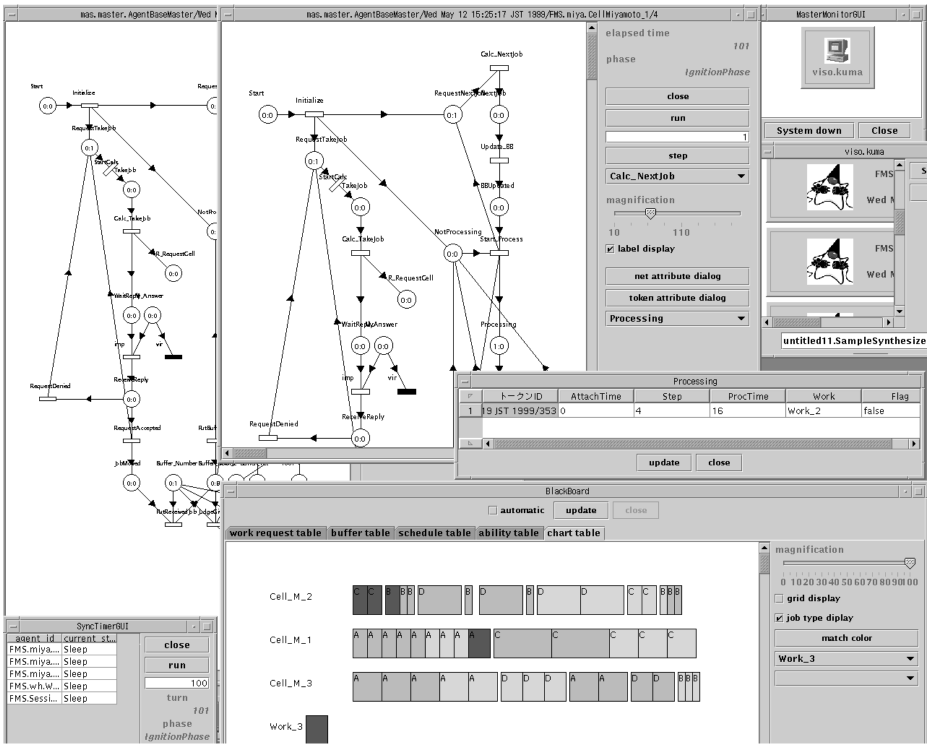


Fig. 5. Agent Nets



mas.master.AgentBaseMaster/Wed May 12 15:25:17 JST 1999/FMS.miya.CellMiyamoto_1/4

elapsed time 101
phase *IgnitionPhase*
1
Calc_NextJob
magnification 10 110
☑ label display
net attribute dialog
token attribute dialog
Processing

MasterMonitorGUI
viso.kuma
untitled11.SampleSynthesize

Processing

	トークンID	AttachTime	Step	ProcTime	Work	Flag
1	19 JST 1999/353	0	4	16	Work_2	false

BlackBoard
☐ automatic

Cell_M_2: C C B B B D B D B D D C C B B B
Cell_M_1: A A A A A A A A A C C C C C
Cell_M_3: A A A A A D D D A A D D B B B
Work_3

magnification 0 10 20 30 40 50 60 70 80 90 100
☐ grid display
☑ job type diplay
Work_3

SyncTimerGUI

agent_id	current_st...
FMS.miya....	Sleep
FMS.miya....	Sleep
FMS.miya....	Sleep
FMS.wh.W...	Sleep
FMS.Sessi...	Sleep

100
turn 101
phase *IgnitionPhase*

Fig. 6. Simulation Screen

– Deciding a next job part. This part decides the next processing work from works in its own buffer (CalcNextJob). The action code with CalcNextJob refers works in BufferBefore, gets information about the works and decide next job by it's own policy.
– Processing the job part. This part process the work which was decided in the deciding a next job part (StartProcess), and put a product into the a buffer. Similar to tokens in the place Moving, tokens in the place Processing have a invalid period. This period means processing time of the work. This time is variable, and we calculate it in the action code of the transition StartProcess.
– Giving jobs part. This part deals with requests from other cells. Transitions JudgeGive or JudgeGive2 judges whether the cell gives the work or not, and answers are sent from RAnswer.
– Sending complete jobs part. This part request the warehouse to take a product from its buffer. Intelligence of the cell agent can be added to the agent net by using programming codes to transitions, places, or arcs, for example, in the action of CalcNextJob. In Fig. 5(a) and Fig. 5(b), agent nets of the upper level and warehouse is shown. Tokens in the upper level net represents agent nets of lower level, e.g., cells and the warehouse. In this case, we have three cell nets in the system. Fig. 6 is the screen dump of the simulation.

5.2 Secret Sharing Distributed Storage System[10]

Figure 7 shows an image of an autonomous distributed storage system. This system is a multi agent system consisting of server agents that reside on storage nodes scattered throughout the network and client agents that receive requests from users. Our objective is to implement a system having a high degree of secrecy, reliability, and robustness according to collaboration between server agents and collaboration between client agents and server agents. Client agents receive user requests, transfer them to server agents, and return the results to users. Client agents can communicate with arbitrary server agents and can switch server agents according to server agent load conditions or the network state. They provide a nonstop service to clients. A data is encrypted by secret sharing method into n numbers of fragments, called as shares, and each share is stored in distributed servers. All server agents have functions for encrypting and decrypting data and for storing and acquiring shares. For decrypting the

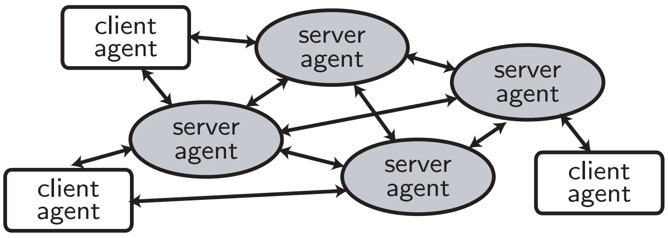


Fig. 7. Distributed Storage System

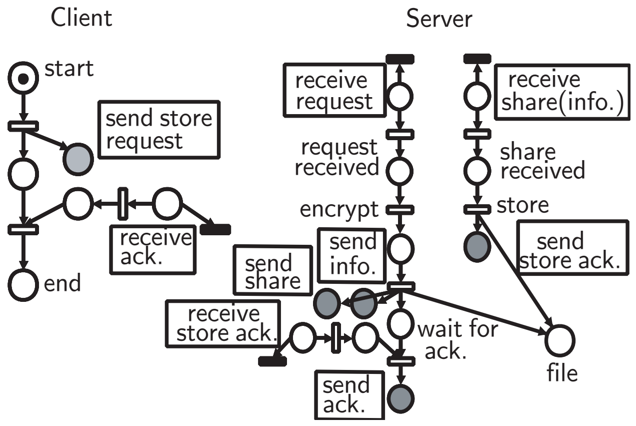


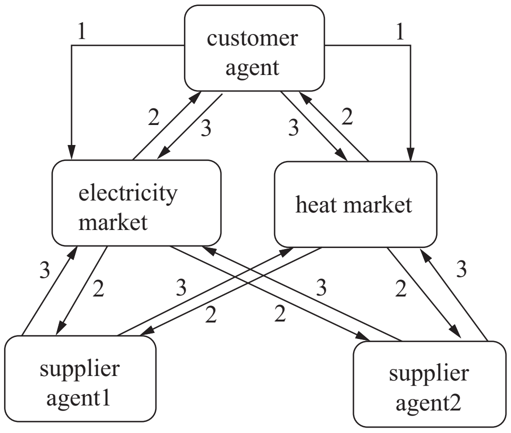
Fig. 8. MAN model for a store process

data, we need k shares out of n shares of the data. If there is a store request from a client, a server agent encrypts the file and sends shares to other server agents based on a storage policy. A server agent that receives a store request saves received shares on a local disk to prepare for future requests. If there is a file fetch request from a client, a server agent collects total k of shares to decrypt the file.

Figure 8 shows an agent net model for a store process. A store task starts by a token being passed from the "send store request" place of the client to the "receive request" place of the server. Encryption is performed due to the firing of an "encrypt" transaction, and the shares and data information are sent from the "send share" and "send info." places to other servers. The other servers receive the shares or data information via the "receive share (info.)" place, and after storing the information in the "file" place, send back an acknowledgement from "send store ack." After the server receives a reception certification of the shares from all servers, it sends back a reception certification to the client, and the store task is completed. Other basic functions of a storage system can also be modeled by MAN for implementing the secret sharing mechanism. The proposed storage system has obvious advantages such as robustness to unexpected disasters and high reliability against data loss or stealing.

5.3 Distributed Energy Management Systems for CO2 Reduction [11]

We consider a problem to minimize both of energy costs and CO2 emissions in an area where there exist several independent entities (agents) who consume and/or produce electric or thermal energy. We adopt a cooperative energy trading decision method instead of total optimization for which usually competitive entities would not like to accept. By this method, we want to reduce energy consumption and the amount of the CO2 emissions at the same time in the entire group without undermining economic



1 : initial bidding condition
2 : update trading price and exhibit bidding price
3 : bid

Fig. 9. Energy Trading System

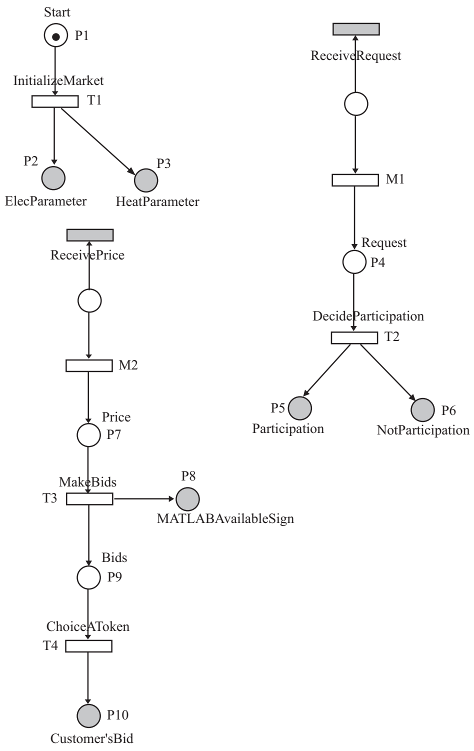


Fig. 10. Customer Agent MAN model

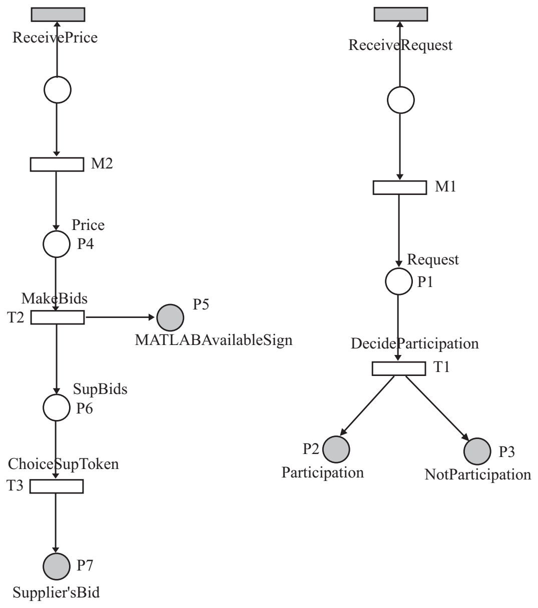


Fig. 11. Supplier Agent MAN model

benefit for each agent. A group consists of agents connected by electrical grids and heat pipelines. Each agent can buy electricity and gas from the electric power company and the gas company outside of the group. Electricity and heat can also be bought and sold between agents in the group. We show an example of a group energy trading system in Fig. 9. The group shown in Fig. 9 has Factory1 and 2 as producer agents and Building as a consumer agent. Each agent has demands for electricity and heat with predicted demand patterns. The producer agents can buy gas and electricity from the outside of the group, can produce electricity and heat to fill their own demands, and can sell energy in the group. The consumer agents can buy energy in the group and from the outside of the group, and can produce energy to fill their own energy demands. Each agent decides its energy purchase plan and running plan of equipments that maximizes its own economic profit that will be subject to energy demands and CO2 emissions. Fig. 10 and Fig. 11 are MAN models for customer agent and supplier agent, respectively. In our energy trading system, not only unit price of energy but CO2 emission basic unit is taken into account. A market-oriented programming (MOP) is a method for constructing a virtual perfect competitive market on computers, making the state of equilibrium which appears as a result of the interaction between agents involved in the market, and deriving the Pareto optimum distribution of goods finally. A market economy considered in the MOP is composed of goods and agents. For formulation of MOP, it is necessary to define (1) goods, (2) agents, and (3) agent’s bidding strategies. In Fig. 12, we show an example of a MAN model of markets in the group shown in Fig. 9.

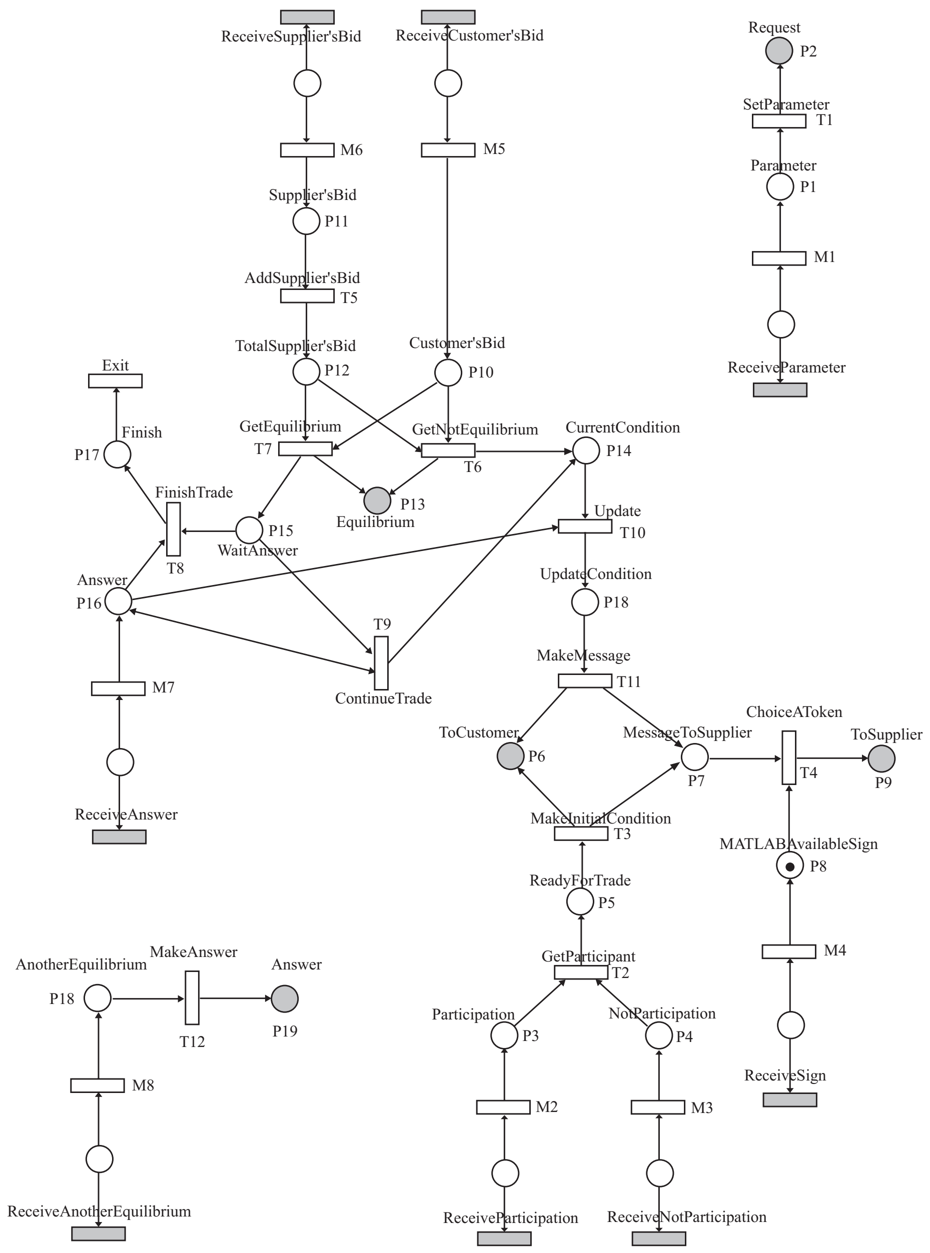


Fig. 12. Market MAN model

6 Conclusion

This paper surveys the works and works-in -progress by the authors and their group on realization of Autonomous Distributed Systems (ADS) by Multi Agent Nets (MAN). ADS is a universal system concept which plays a central role in this ubiquitously networked society and the realization technologies are so vital for safe, robust, and flexible management and operation of such systems. We proposed MAN as one of the basic modeling and prototyping tools and several applications revealed its versatility for wide range of engineering and business problems. On a theoretical or formal verification analysis on MAN, there remains serious works yet to be done. An attempt to generate an abstract state space for MAN and a verification by ACTL on the abstract space is a few example of the formal treatment for nested Petri nets like MAN [12]. The use of MAN as an executable verification model for high level modeling language such as UML is also promising and the construction of simulator for UML based Service Oriented Architecture of business process is undergoing by this group.

Acknowledgement

Taking an advantage of being an invited paper, the authors made a survey of their own ideas and the references are subjective and not complete. The authors would like to express their sincere thanks to numbers of graduate students, especially to Mr. Y.Morihiro, who made a joint research project described throughout in this paper. He also thanks to Program Committee chairs, Drs. Jetty Kleijn and Alex Yakovlev for giving us the opportunity to prepare this paper.

References

1. Lee, D.I., Kumagai, S., Kodama, S.: Complete Structural Characterization of State Machine Allocatable Nets. IEICE Trans. E74(10), 3115–3123 (1991)
2. Lee, D.I., Kumagai, S., Kodama, S.: Handles and Reachability Analysis of Free Choice Nets. In: DeMichelis, G., Díaz, M. (eds.) Application and Theory of Petri Nets 1995. LNCS, vol. 935, pp. 298–315. Springer, Heidelberg (1995)
3. Blanc, C.S.: Cooperative Nets. In: Proc. of Object-Oriented Programming and Models of Concurrency (1995)
4. Bastide, R.: Approaches in Unifying Petri Nets and the Object-Oriented Approach. In: Proc. of Object-Oriented Programming and Models of Concurrency (1995)
5. Miyamoto, T., Kumagai, S.: A Survey of Object-Oriented Petri Nets and Analysis Method, Invited Paper. IEICE Trans. E88-A (11), 2964–2971 (2005)
6. Moldt, D., Valk, R.: Business Process Management. LNCS, vol. 1806, pp. 254–273. Springer, Heidelberg (2000)
7. Miyamoto, T., Kumagai, S.: A Multi Agent Net Model of Autonomous Distributed Systems. In: Proc. of CESA'96, Symposium on Discrete Event and Manufacturing Systems, pp. 619–623 (1996)
8. Miyamoto, T., Kumagai, S.: A Multi Agent Net Model and the realization of Software Environment. In: Workshop Proc. of Application of Petri Nets to Intelligent System Development, 20th ICATPN, pp. 83–92 (1999)

9. Miyamoto, T., Ichimura, D., Kumagai, S.: A Multi Agent Based Manufacturing Resource Planning and Task Allocation Systems. IEICE Trans. E86-A (4), 806–812 (2003)
10. Miyamoto, T., Doi, S., Nogawa, H., Kumagai, S.: Autonomous Distributed Secret Sharing Storage Systems. Syst. and Comp. in Jpn 37(6), 55–63 (2006) (Wiley Periodicals, Selected English Translation from IEICE Trans. J87-D-I(10), 899–906, 2004)
11. Yakire, K., Miyamoto, T., Kumagai, S., Mori, K., Kitamura, S., Yamamoto, T.: An Energy Distribution Decision Method in Distributed Energy Management Systems by the Market-Oriented Programming. In: Proc. of SICE-ICCAS International Joint Conference 2006, Busan, Korea (2006)
12. Miyamoto, T., Kumagai, S.: On the reachability Analysis of Multi Agent Nets. In: Proc. of SICE Annual Conference of Systems and Information, pp. 247–250, (November 2006)

Petri Nets Without Tokens
(Invited talk)

Antoni Mazurkiewicz

Institute of Computer Science of PAS
Ordona 21, 01-237 Warsaw, Poland
Antoni.Mazurkiewicz@ipipan.waw.pl

For more than 40 years Petri nets [1] serve as an efficient formal model of concurrent behavior of complex discrete systems. There exists a rich bibliography of books and works devoted to these methods and many applications of nets have been already created. The model is extremely simple: it uses three basic concepts, of places, of transitions, and of a flow relation. The behavior of a net is represented by changing distribution of tokens situated in the net places, according to some simple rules (so-called firing rules). Non-negative integers play essential role in the description of tokens distribution, indicating the number of tokens contained in nets places and . Transitions determine way of changing the distribution, taking off a number of tokens from entry places and putting a number of tokens in exit places of an active transition. Formally, to any transition some operations on numbers stored in places are assigned and therefore the behavior of nets is described by means of a simple arithmetic with adding or subtracting operations on non-negative integers.

Nets became attractive for several reasons, namely because:

- simplicity of description they offer,
- demonstrativeness, mainly due to their graphic representation
- a deep insight into concurrency phenomena,
- facility of applications to basic concurrent systems

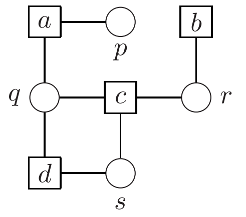
However, some of positive features of nets create also their weakness. Namely, simplicity of the model makes difficult the complex situations description; the formalism being well suited to describe basic phenomena may turn out to be difficult for some real applications. Integers may turn out to be too primitive structures for dealing with more subtle objects; enlarging the net structure to the real word situations description could be troublesome. Finally, the intuition appealing graphical representation may be dangerous for rigorous formal reasoning. For these reasons, a constantly growing number of different extensions of original nets to the so-called higher level ones has been invented. In this talk a very modest modification of the original net concept is discussed, consisting in:

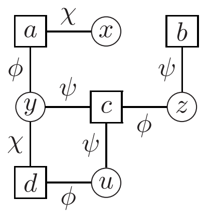
- replacing integers by arbitrary objects for storing in places;
- accepting arbitrary relations as transition activities;
- resigning from a fixed flow direction in the net structure.

Token-free nets. A token-free net (FT-net, for short) is defined by its *structure* and *specification*; the structure is defined by triple (P, T, γ), where P, T are finite, non-empty disjoint sets, of *places* and *transitions* respectively, and $\gamma \subseteq P \times T$ is a relation called the *neighborhood* relation. Set

$$\gamma(t) = \{p \mid (p,t) \in \gamma\}, \qquad \gamma(p) = \{t \mid (p,t) \in \gamma\}.$$

Say that transitions t', t'' are *independent* and write $(t', t'') \in I$, if $\gamma(t') \cap \gamma(t'') = \emptyset$. The specification of net with structure (P, T, γ) is any triple (S, F, σ^0), where S is a mapping which to each place $p \in P$ assigns set S_p (of *values* of place p), F is a mapping which to each $p \in P$ and each $t \in \gamma(p)$ assigns binary relation $F_p(t) \subseteq S_p \times S_p$ (the *transition* relation), and σ_0 is a mapping which to each $p \in P$ assigns value $\sigma^0(p) \in S_p$ (the *initial valuation* of places). Below, in the left-hand picture, a TF-net with structure $(\{p,q,r,s\}, \{a,b,c,d\}, \{(p,a), (q,a), (q,c), (q,d), (r,b), (r,c), (s,c), (s,d)\})$ is presented in a graphical form. In the right-hand picture this structure is filled up with a specification, giving some initial values x, y, z, u assigned to places and transition relations ϕ, χ, ψ assigned to corresponding edges, e.g. $F_p(a) = \chi$, $F_q(c) = \psi$, $F_q(d) = \chi$.





Let (S, F, σ^0) be a specification of net N with structure (P, T, γ). Any mapping σ assigning to place $p \in P$ its value $\sigma(p) \in S_p$ is called a *valuation*. Transition t is *enabled* at valuation σ, if $\sigma(p)$ is in the domain of $F_p(t)$ for any neighbor place p of t. The behavior of N is defined as follows. Let $\sigma', \sigma'' \in P \longrightarrow S$ be two valuations of places of N, $t \in T$ be a transition of N; say that t transforms in N valuation σ' into σ'' (or that t can *fire* at valuation σ') and write $\sigma' \xrightarrow{t}_N \sigma''$, if the following equivalence holds:

$$\sigma' \xrightarrow{t}_N \sigma'' \quad \Leftrightarrow \quad \left| \begin{array}{ll} (\sigma'(p), \sigma''(p)) \in F_p(t), & \forall p \in P_t, \\ \sigma''(p) = \sigma'(p), & \forall p \notin P_t. \end{array} \right.$$

From this definition it follows that $\sigma' \xrightarrow{t}_N \sigma''$ implies t to be enabled at $\sigma'(p)$. Extend relation $\xrightarrow{t}_N$ to $\xrightarrow{w}_N$ for $w \in T^*$ in the standard way:

$$\sigma' \xrightarrow{w}_N \sigma'' \Leftrightarrow \left| \begin{array}{l} \sigma' = \sigma'', \text{ if } w = \epsilon, \\ \exists \sigma : s'\sigma' \xrightarrow{u}_N \sigma \xrightarrow{t}_N \sigma'', \text{ if } w = ut, u \in T^*, t \in T. \end{array} \right.$$

The set $\mathrm{Seq}(N) = \{w \mid \exists \sigma : \sigma^0 \xrightarrow{w}_N \sigma\}$ is called the (sequential) *behavior* of N, and its elements, traditionally, *firing sequences* of N.

Classical place/transition nets are particular cases of TF-net, in which values of all places are non-negative integers, and relations are either defined by $F_p(t) = \{n, n+k) \mid n+k \leq c\}$ (the pair (p,t) is then represented by an arrow leading from t to p labeled with k, the *multiplicity* of that arrow, and with place p labeled with c, the capacity of p), or by $F_p(t) = \{n, n-k) \mid k \leq n\}$ (then the arrow, similarly labeled with k, leads from p to t). The behavior of such nets agrees with the definition given above.

As another example of a TF-net can serve 'string nets', with strings over alphabet Σ are individual values, and relations are either defined by $F_p(t) = \{(w, wa) \mid w \in \Sigma^*\}$ or by $F_p(t) = \{(wa, w) \mid w \in \Sigma^*\}$, for some $a \in \Sigma$, or by $F_p(t) = \{(w, w) \mid \epsilon \neq w \in \Sigma^*\}$. Observe also that values assigned to different places may have different types, e.g. values of some places can be strings, while those of other ones can be integers.

Composition properties. TF-nets enjoy a composition property similar to that holding by classical Petri nets [2]. Let $N = (P, T, \gamma; S, \boldsymbol{F}, \sigma^0)$ be a TF-net. To each place $p \in P$ assign finite automaton A_p defined by equality $A_p = (S, T_p, F_p, s_p^0)$, called the *local automaton* for p in N, where $T_p = \gamma(p), F_p(t) \subseteq S \times S$, and $s_p^0 = \sigma^0(p)$. Conversely, given a finite family of local automata $A_i = (S_i, T_i, F_i, s_i^0)$, $i \in I$, with $F_i(t) \subseteq S_i \times S_i$ for each $t \in T_i$ and $s_i^0 \in S_i$, their composition $\&_{i \in I} A_i$ can be defined as TF-net $(I, T, \gamma; S, F, \sigma^0)$, where $T = \bigcup_{i \in I} T_i, \gamma = \{(i,t) \mid i \in I, t \in T_i\}$, $S(i) = S_i$ for $i \in I$, and $\sigma^0(i) = s_i^0$. Extend $F_p(t)$ to $F_p^*(w)$ for all $w \in T^*$:

$$(s', s'') \in F_p^*(\epsilon) \Leftrightarrow s' = s'',$$
$$(s', s'') \in F_p^*(wt) \Leftrightarrow \exists s : (s', s) \in F_p^*(w) \wedge (s, s'') \in F_p(t).$$

for all $s', s'' \in S, w \in T^*, t \in T$. The language accepted by A_p is the set $L(A_p) = \{w \mid \exists s : (s_p^0, s) \in F^*(w)\}$. We have the following composition property:

$$\mathrm{seq}(\&_{i \in I} A_i) = \ker(\&_{i \in I} L(A_i)),$$

where $w \in \&_{i \in I} L(A_i) \Leftrightarrow \forall i \in I : \pi(w, T_i) \in L(A_i)$, $\pi(w, T_i)$ is the projection of w onto T_i, and $\ker(L)$ is the greatest prefix closed subset of language L. A set of strings is *directed*, if any two elements of this set are prefixes of another element of this set. Any maximum prefix-closed directed subset of $\mathrm{seq}(N)$ is a full *run* of N; the set of all full runs of N is denoted by $\mathrm{Seq}(N)$. Cardinality of a full sequential run (possibly infinite) is called its length.

Net restrictions. Flexibility of choosing arbitrary transition relations as specification of nets with a common structure offers a possibility of comparison such specifications. Let N', N'' be nets with the same structure, say (P, T, γ), and with specifications $(S', F', \sigma_1^0), (S'', F'', \sigma_2^0)$, respectively. Say that net N' is a *restriction* of net N'', if for each $p \in P$ there exists mapping $\delta_p : S'_p \longrightarrow S''_p$ such that

$$s_2^0(p) = \delta_p(s_1^0(p)), \qquad (s', s'') \in F'_p(t) \Rightarrow (\delta_p(s'), \delta_p(s'')) \in F''_p(t).$$

and then write $N' \leq N''$. From the definition of nets behavior it follows easily

$$N' \leq N'' \Rightarrow \text{seq}(N') \subseteq \text{seq}(N'').$$

The case when δ_p is the identity mapping for each $p \in P$ is a particular case of restriction; then the restriction condition reduces to the inclusion $F'_p(t) \subseteq F''_p(t)$ for all $p \in P, t \in T$, i.e. simply to $F' \subseteq F''$. Clearly, the restriction relation is a partial order in the family of all nets with a common structure; the net with $F_p(t) = \emptyset$ (over-specified net, where none of transitions are enabled at any valuation) is its minimum element. In a net with $\emptyset \neq F_p(t) = S_p \times S_p$ any transition at any valuation is enabled (under-specified net), is maximum element in the above ordering. The question arises if it is possible to adjust the specification of a net to get a priori chosen behavior of it. A partial answer to this question is formulated below.

Restricted nets behavior. Let $N = (P, T, \gamma; S, F, \sigma^0)$ be an arbitrary but fixed TF-net. Let $\equiv$ be the least equivalence relation in T^* s.t. $(t', t'') \in I \Rightarrow t't'' \equiv t''t'$ and $w' \equiv w'', u' \equiv u'' \Rightarrow w'u' \equiv w''u''$; then it follows easily that $w' \in \text{Seq}_N \wedge w' \equiv w'' \Rightarrow w'' \in \text{Seq}_N$. It is also not difficult to show that two strings $w', w'' \in \text{seq}(N)$ are equivalent if and only if $\forall p \in P : \pi(w', \gamma(p)) = \pi(w'', \gamma(p))$ (their projections on local alphabets are pairwise equal). Extend equivalence relation $\equiv$ to sets of strings by the equivalence:

$$W' \equiv W'' \Leftrightarrow \forall p \in P : \{\pi(w, \gamma(p)) \mid w \in W'\} = \{\pi(w, \gamma(p)) \mid w \in W\text{''}\}.$$

This description method origins from basic results of Shields [3]. Possible effects of restrictions are demonstrated by the following fact. Let $W_0 \in \text{Seq}(N)$. Then there exists a restriction N' of N such that any run $W' \in \text{Seq}(N')$ is equivalent to W:

$$\forall\ W_0 \in \text{Seq}(N) : \exists\ N' \leq N : \forall\ W \in \text{Seq}(N') : W \equiv W_0.$$

This result is a partial answer to the more general question about defining possibilities of specification restrictions.

References

[1] Petri, C.A.: Concepts of Net Theory. In: Proc. of MFCS'73, High Tatras, Math.Institut of Slovak Academy of Sciences, pp. 137–146 (1973)

[2] Mazurkiewicz, A.: Semantics of Concurrent Systems: A Modular Fixed Point Trace Approach, Instituut voor Toegepaste Wiskunde en Informatica, Rijksuniversiteit Leiden, TR-84-19 (1984)

[3] Shields, M.W.: Non-sequential behaviour, part I. Int. Report CSR-120-82, Dept. of Computer Science, University of Edinburgh (1979)

Toward Specifications for Reconfigurable Component Systems[⋆]
(Preliminary Abstract)

Andrzej Tarlecki

Institute of Informatics, Warsaw University
and Institute of Computer Science PAS, Warsaw, Poland

Serious developments concerning formal aspects of software specification, verification and development initially addressed programming-in-the-small: description, analysis and verification of relatively simple (iterative) programs. This phase was marked with achievements like Hoare's logic [Hoa69], formalising the earlier proposals concerning proofs of correctness of iterative programs. Early work of Parnas [Par72b, Par72a] advanced programming-in-the-large, undertaking the issues of modularity and abstraction. Data abstraction together with the related issue of step-wise program development, was addressed in another seminal paper by Hoare [Hoa72].[1] Work on algebraic specifications (e.g. [GTW78]) initially addressed the problem of specifying (abstract) data types, but soon grew to a general methodology for specification and systematic development of software system with an extensive body of work and publications [BKL+91]. Our relatively recent account of one line of work grown on algebraic specifications given in [ST97, Tar03] presents foundations for a comprehensive methodology for software specification and development, and rightly claims generality of the approach gained by parameterising on an arbitrary logical system formalised as an institution [GB92]. Nevertheless, the approach presented there works most clearly for relatively simple systems built by statically combining well-specified, hierarchical modules. This is perhaps best visible in particular specification formalisms that fit this approach, with CASL [BM04, CoF04] developed recently by the CoFI group as a prime example. Oversimplifying grossly: we know best how to specify and put together purely functional STANDARD ML modules and their relatively minor variants (even if the EXTENDED ML experiment [KST97] with embedding specifications into arbitrary STANDARD ML programs was only partly successful, it certainly does work for specifications of such systems).

In the meantime, the standard practise of building complex software system changed considerably, for better or worse encompassing many complex programming features with often far from satisfactory semantic and logical tools for their description and analysis, in spite of considerable amount of work and a whole

⋆ This work was funded in part by the European IST FET programme under the IST-2005-015905 MOBIUS and IST-2005-016004 SENSORIA projects.

[1] I make no pretence of any completeness or even full accuracy of such historical references here — instead, let me advertise excellent essay [Jon03] on the history of approaches to, and methods for verification of (sequential imperative) programs.

spectrum of approaches proposed. For instance, we have object-oriented technology [Boo94], with local states of objects, inheritance and dynamic communication between objects, to mention just a few features not treated directly by specification formalisms like CASL. Of course, this does not mean that in principle the object-oriented paradigm cannot be dealt with using the general methodology referred to above. In particular, many aspects of object-oriented paradigm are covered by behavioural interpretation of specifications [Rei86, GM00, BH06, BST06], where internal features of a specification (including for instance hidden data to model the state) are regarded only in so far as they influence the visible behaviour of the specified system. Still, the practise of object-oriented design and development is dominated by approaches like UML [BRJ98], in spite of its deficiencies and (at least initially) lack of formal foundations. One aspect of UML which we want to stress here is that via dozens of kinds of UML diagrams, it offers a considerable heterogeneous spectrum of methods to capture various facets of the specified systems.

The methodology advocated in [ST97] and [BM04] naturally leads to the development of systems that are statically composed of a number of mutually independent, well-specified modules. This is quite orthogonal to the current developments in the area of, for instance, service-oriented computing, where systems are configured dynamically out of a collection of available components to provide a given service (and often decomposed once the desired goal has been achieved). This dynamic aspect is to some extent present for instance in agent-oriented technologies (e.g. [Woo01]). It seems though that the stress there is on the search for the right services offered, not so much on the right specification of the modules that provide the services and on the systematic design of the overall architectures of such systems.

What I would like to pursue is an approach to specification, design and systematic development of hierarchical systems that consist of a collection of components designed to be combined in a number of possible ways to potentially fulfil various tasks. It should be possible to reconfigure the resulting systems dynamically using the components provided. The framework should enable subsequent specification of individual components, their alternative configurations, the overall system capabilities and properties ensured by particular configurations, as well as the possible system reconfiguration process itself. Moreover, the framework aimed at should be hierarchical in the sense that overall system specification should have the same format as specifications of individual components, so that the components can also be built as such reconfigurable (sub-)component systems.

This is, so far, a long term goal with not much specific proposals on how to reach it in any formal and mathematically well-founded way. I would like, however, to present at least some directions of work and ideas which in my view are useful prerequisites to achieve such a goal.

Specifications in an Arbitrary Institution. Since the early work on CLEAR [BG80], the formalisation of logical systems as *institutions* [GB84, GB92] proved to be not only a powerful conceptual basis for a very abstract version of model theory, but first of all, a pragmatically useful technical tool in the design of specification formalisms and in the work on foundations of software specification and systematic development. This started with mechanisms to structure specifications built in any logical system presented as an institution [ST88a], and then covered a whole spectrum of standard concepts and ideas in the area [ST97, Tar03].

Systematic Development and Architectural Specifications. Starting with [ST88b], the work on fundamentals of specification theory in the framework of an arbitrary institution included a general foundational view of the systematic development of correct modular systems from their requirements specifications. This led to formalisation of the design steps in software development process as *architectural specifications* [BST02], whereby a modular structure of the system being developed is designed by listing the units the system is to be built of, providing their precise specifications, and defining the way in which they are supposed to be composed to build the overall system.

Heterogeneous Logical Environments. The theory of institutions also offers solid foundations to deal with *heterogeneous specifications* [Tar96, Tar00, Mos02], where specifications of various modules of a given system, or various facets of some modules may be conveniently built in various logical systems, most appropriate for the particular task at hand. One related idea is that of *heterogeneous development*, where the process of the system development may migrate from one logical system to another, most appropriate at the given stage. Of course, to make this meaningful, the logical systems in use, or rather the institutions that capture them, must be linked with each other by a map of one kind or another [GR02], thus forming a *heterogeneous logical environment*, which in turn may be formalised simply as a diagram in a category of institutions.

Architectural Design and Connectors. Architectural specifications mentioned above provide a tool for designing a static modular structure of the software system under development. This is not quite the same as the overall system architecture, as discussed for instance in [AG97], which deals more with the actual interconnection between system components in the dynamic process of computations carried out by the system. One crucial idea there is that in the architectural design the system components need not be linked directly with each other, but rather via *architectural connectors* [FLW03] that are specialised units playing no other role than to coordinate activity of the components they link.

Systems of Potentially Interconnected Components. What emerges from the above is a view of systems as collections of components linked with each other by architectural connectors. One observation in [Zaw06] is that the resulting collection of interconnected components as a whole may be of a very limuited use, or may even be inconsistent due to potentially contradictory properties of actions offered by various components, while its various sub-configurations may amount to perfectly useful systems. As a consequence we can meaningfully consider systems of components with their *potential interconnections* from which only some are “active” at a given moment. Various techniques, for instance those based on graph grammars, may now be used to reconfigure the actual "active" system within the possibilities offered by the graph of potential connection between system components.

The above list indicates the ideas, concepts and results that can be put together and further developed aiming at foundations for an overall methodology of specification and development of hierarchical component systems that allow for system reconfiguration. Preliminary combination of the above ideas and initial results will be presented at the conference, with a hope to generate a critical discussion and suggestions of the techniques developed within the field of Petri nets that may be useful in this context. Undoubtedly though, achieving the overall goal requires considerable further work.

References

[AG97] Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)

[BG80] Burstall, R.M., Goguen, J.A.: The semantics of Clear, a specification language. In: Bjorner, D. (ed.) Abstract Software Specifications, 1979 Copenhagen Winter School, LNCS, vol. 86, pp. 292–332. Springer, Heidelberg (1980)

[BH06] Bidoit, M., Hennicker, R.: Constructor-based observational logic. Journal of Logic and Algebraic Programming 67(1-2), 3–51 (2006)

[BKL$^+$91] Bidoit, M., Kreowski, H.-J., Lescanne, P., Orejas, F., Sannella, D. (eds.): Algebraic System Specification and Development. LNCS, vol. 501. Springer, Heidelberg (1991)

[BM04] Chapters by Mossakowski, T., Sannella, D., Tarlecki, A. In: Bidoit, M., Mosses, P.D. (eds.): CASL User Manual. LNCS, vol. 2900 Springer, Heidelberg (2004)

[Boo94] Booch, G.: Object-Oriented Analysis and Design with Applications, 2nd edn. Addison-Wesley, London (1994)

[BRJ98] Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, London (1998)

[BST02] Bidoit, M., Sannella, D., Tarlecki, A.: Architectural specifications in Casl. Formal Aspects of Computing 13, 252–273 (2002)

[BST06] Bidoit, M., Sannella, D., Tarlecki, A.: Observational interpretation of Casl specifications. Research Report LSV-06-16, Laboratoire Spécification et Vérification, ENS Cachan (Submitted for publication 2006)

[CoF04] CoFI (The Common Framework Initiative). Casl Reference Manual. LNCS 2960 (IFIP Series). Springer (2004)

[FLW03] Fiadeiro, J.L., Lopes, A., Wermelinger, M.: A mathematical semantics for architectural connectors. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming. LNCS, vol. 2793, pp. 178–221. Springer, Heidelberg (2003)

[GB84] Goguen, J.A., Burstall, R.M.: Introducing institutions. In: Clarke, E., Kozen, D. (eds.) Logics of Programs. LNCS, vol. 164, pp. 221–256. Springer, Heidelberg (1984)

[GB92] Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. Journal of the ACM 39(1), 95–146 (1992)

[GM00] Goguen, J.A., Malcolm, G.: A hidden agenda. Theoretical Computer Science 245(1), 55–101 (2000)

[GR02] Goguen, J.A., Rosu, G.: Institution morphisms. Formal Aspects of Compututing 13(3-5), 274–307 (2002)

[GTW78] Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types. In: Yeh, R. (ed.) Current Trends in Programming Methodology, IV, pp. 80–149. Prentice-Hall, Englewood Cliffs (1978)

[Hoa69] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–583 (1969)

[Hoa72] Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1(4), 271–281 (1972)

[Jon03] Jones, C.B.: The early search for tractable ways of reasoning about programs. IEEE Annals of the History of Computing 25(2), 26–49 (2003)

[KST97] Kahrs, S., Sannella, D., Tarlecki, A.: The definition of Extended ML: A gentle introduction. Theoretical Computer Science 173, 445–484 (1997)

[Mos02] Mossakowski, T.: Heterogeneous development graphs and heterogeneous borrowing. In: Nielsen, M., Engberg, U. (eds.) ETAPS 2002 and FOSSACS 2002. LNCS, vol. 2303, pp. 326–341. Springer, Heidelberg (2002)

[Par72a] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)

[Par72b] Parnas, D.L.: A technique for software module specification with examples. Communications of the ACM 15(5), 330–336 (1972)

[Rei86] Reichel, H.: Behavioral program specification. In: Poigné, A., Pitt, D.H., Rydeheard, D.E., Abramsky, S. (eds.) Category Theory and Computer Programming. LNCS, vol. 240, pp. 390–411. Springer, Heidelberg (1986)

[ST88a] Sannella, D., Tarlecki, A.: Specifications in an arbitrary institution. Information and Computation 76, 165–210 (1988)

[ST88b] Sannella, D., Tarlecki, A.: Toward formal development of programs from algebraic specifications: Implementations revisited. Acta Informatica 25, 233–281 (1988)

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[Tar96] Tarlecki, A.: Moving between logical systems. In: Haveraaen, M., Dahl, O.-J., Owe, O. (eds.) Recent Trends in Data Type Specification, ADT'95. LNCS, vol. 1130, pp. 478–502. Springer, Heidelberg (1996)

[Tar00] Tarlecki, A.: Towards heterogeneous specifications. In: Gabbay, D., de Rijke, M. (ed.) Frontiers of Combining Systems 2, Studies in Logic and Computation, pp. 337–360. Research Studies Press (2000)

[Tar03] Tarlecki, A.: Abstract specification theory: An overview. In: Broy, M., Pizka, M. (eds.) Models, Algebras, and Logics of Engineering Software. NATO Science Series — Computer and System Sciences, vol. 191, pp. 43–79. IOS Press, Amsterdam (2003)

[Woo01] Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley, Chichester (2001)

[Zaw06] Zawłocki, A.: Diagram models of interacting components. Technical report, Institute of Informatics, Warsaw University (2006)

Generating Petri Net State Spaces

Karsten Wolf

Universität Rostock, Institut für Informatik,
18051 Rostock, Germany
karsten.wolf@informatik.uni-rostock.de

Abstract. Most specific characteristics of (Place/Transition) Petri nets can be traced back to a few basic features including the *monotonicity* of the enabling condition, the *linearity* of the firing rule, and the *locality* of both. These features enable "Petri net" analysis techniques such as the invariant calculus, the coverability graph technique, approaches based on unfolding, or structural (such as siphon/trap based) analysis. In addition, most verification techniques developed outside the realm of Petri nets can be applied to Petri nets as well.

In this paper, we want to demonstrate that the basic features of Petri nets do not only lead to additional analysis techniques, but as well to improved implementations of formalism-independent techniques. As an example, we discuss the explicit generation of a state space. We underline our arguments with some experience from the implementation and use of the Petri net based state space tool LoLA.

1 Introduction

Most formalisms for dynamic systems let the system state evolve through reading and writing variables. In contrast, a Petri net marking evolves through the consumption and production of resources (tokens). This fundamental difference has two immediate consequences. First, it leads to a *monotonous enabling condition*. This means that a transition enabled in some state is as well enabled in a state that carries additional tokens. Second, it leads to the *linearity of the firing rule*. This means that the effect of a transition can be described as the addition of an integer vector to a marking vector.

The Petri net formalism has another fundamental property that it shares with only some other formalisms: *locality*. This means that every transition depends on and changes only few components of a state. In Petri nets, these components (places) are explicitly visible through the arc (flow) relation.

Many specific Petri net analysis techniques can be directly traced back to some of these characteristic features of Petri nets. For instance, the *coverability graph* generation [KM69,Fin90] is closely related to monotonicity and linearity. The *invariant* calculus [LS74,GL83,Jen81] clearly exploits linearity of the firing rule. For structural analysis such as *Commoner's Theorem* [Com72] or other methods based on siphons and traps, monotonicity and locality may be held responsible. Petri net *reduction* [Ber86] is based on the locality principle. Analysis based on *unfoldings* [McM92,Esp92] (or partial ordered runs) involves the locality principle as well. This list could be continued.

On the other hand, there are several verification techniques that do not depend on a particular modeling language. An example for this class of techniques is *state space* based analysis, and—build on top of it—*model checking* [CE81,QS81]. The availability of specific techniques like the ones mentioned above has long been observed as a particular advantage of Petri nets as a modeling language. Beyond this insight, we want to discuss the implementation of a Petri net based state space generator (LoLA [Sch00c]) in order to show that the basic features of Petri nets mentioned in the beginning of this section can be used for an efficient implementation of—otherwise formalism-independent—explicit state space generation.

We start with a short introduction to the tool LoLA and give, in Sec. 3 a brief overview on some case studies conducted with this tool. In Sec. 4, we consider the implementation of fundamental ingredients of a state space generator such as firing a transition or checking for enabledness. Finally, in Sec. 5, we informally discuss a few Petri net specific issues of state space reduction techniques.

2 The State Space Generation Tool LoLA

The LoLA project started in 1998. Its original purpose was to demonstrate the performance of state space reduction techniques developed by the author. Most initial design decisions were driven by prior experience with the state space component of the tool INA [RS98].

LoLA can read place/transition nets as well as high-level nets. High-level nets are immediately unfolded into place/transition nets So, LoLA is indeed a place-transition net tool while the high-level input can be seen as a shorthand notation for place/transition nets.

LoLA generates a state space always for the purpose of verifying a particular property. It implements the in-the-fly principle, that is, it stops state space generation as soon as the property being verified is determined. The properties that can be verified include

– Reachability of a given state or a state that satisfies a given state predicate;
– Boundedness of the net or a given place;
– Quasiliveness of a given transition;
– Existence of deadlocks;
– Reversibility of the net;
– Existence of home state;
– Liveness of a state predicate;
– Validity of a CTL formula;
– Validity of the LTL formulas $F\phi$, $GF\phi$, and $FG\phi$, for a given state predicate ϕ.

LoLA can build a state space in depth-first or breadth-first order, or it can just generate random firing sequences for an incomplete investigation of the state space. While depth-first search is available for the verification of all properties listed above, the other search methods are restricted to reachability, deadlocks and quasiliveness.

The following reduction techniques are available in LoLA:

– Partial order reduction (the stubborn set method);
– The symmetry reduction;
– The coverability graph generation;
– The sweep-line method;
– Methods involving the Petri net invariant calculus;
– a simplified version of bit hashing

The techniques are employed only if, and in a version that, preserves the property under investigation. If applicable, reduction techniques can be applied in combination.

Using LoLA amounts to executing the following steps.

1. Edit a particular file `userconfig.H` in the source code distribution for selecting the class of property to be verified (e.g., "boundedness of a place" or "model check a CTL formula") and the reduction techniques to be applied.
2. Translate the source code into an executable file `lola`.
3. Call `lola` with a file containing the net under investigation, and a file specifying the particular instance of the property (e.g., the name of the place to be checked for boundedness, or the particular CTL formula to be verified).
4. LoLA can return a witness state or path, an ASCII description of the generated state space, and some other useful information.

Instead of a stand alone use of LoLA, it is also possible to rely on one of the tools that have integrated LoLA, e.g.,

– CPN-AMI [KPA99],
– The Model Checking Kit [SSE03]
– The Petri net Kernel [KW01]

3 Some Applications of LoLA

As LoLA can be downloaded freely, we do not have a complete overview on its applications. In this section, we report on a few case studies conducted by ourselves, and studies we happen to know about.

Validating a Petri Net Semantics for BPEL [HSS05]

WS-BPEL (also called BPEL or BPEL4WS, [Cur+03]) is an XML-based language for the specification of web services. Due to an industrial standardization process involving several big companies, the language contains a lot of non-orthogonal features. It was thus adequate to give a formal semantics to BPEL in order to reason about consistency and unambiguity of the textual specification. Several formal semantics have been proposed in various formalisms, among them two which are based on Petri nets. One of the Petri net semantics has been reported in [HSS05]. It translates every language construct of BPEL into a Petri net fragment. The fragments are glued together along the syntactical structure of a BPEL specification. As the fragments interact in a nontrivial way, it was necessary to validate the semantics. The validation was done by translating several BPEL specifications into Petri nets and letting LoLA verify a number of crucial properties concerning the resulting nets.

The most successful setting in the application of LoLA was the combination of partial order reduction (stubborn sets) with the sweep-line method. Partial order reduction performed well as BPEL activities can be executed concurrently (availability of a significant number of concurrent activities is essential for the partial order reduction). Furthermore, the control flow of a BPEL specification follows a pattern of progress towards a fixed terminal configuration. Such progress can be exploited using the sweep-line method. LoLA detects the direction of progress automatically (this issue is further discussed in Sec. 5).

It turned out that LoLA was able to solve verification tasks for services with up to 40 BPEL activities while it ran out of memory for a service with little more than 100 activities. The nets that could be successfully verified consisted of about 100 to 400 places and 250 to 1.000 transitions. Reduced state spaces had up to 500.000 states. With the capability of handling BPEL specifications with 40 or 50 activities, LoLA is well in the region of practical relevance. So, LoLA has become part of a tool chain for web services that is developed within the Tools4BPEL project [T4B07]. There, ideas exist to tackle larger processes through further tuning the translation from BPEL to Petri nets, and through abstraction techniques to be applied prior to state space verification.

The main lesson learned of this application was that partial order reduction in combination with the sweep-line method is a powerful combination of reduction techniques in the area of services.

Detecting Hazards in a GALS Circuit [SRK05]

GALS stands for globally asynchronous, locally synchronous circuits. A GALS design consists of a number of components. Each component has its own clock signal and works like a synchronous circuit. Communication between components is organized in an asynchronous fashion. This way, designers try to save the advantages of a synchronous design (tool support, clearly understood setting) while they tackle the major drawbacks (speed limitation and high energy consumption due to long distance clock signal paths, energy consumption in idle parts of the circuit).

In collaboration with the Institute of Semiconductor Physics in Frankfurt/-Oder, we investigated a GALS circuit for coding and decoding data of the 802.11 wireless LAN protocol. In particular, we translated a so-called GALS wrapper gate by gate into a place/transition net. A GALS wrapper is the part of a GALS design that encapsulates a component, manages incoming data, and pauses the clock of the component during periods where no data are pending. Consequently, a wrapper as such is an asynchronous circuit. It consists of five components: an input port maintaining incoming data, an output port managing outgoing data, a timeout generator, a pausable clock, and a clock control. All in all, there are 28 gates (logical gates, flip-flops, muller-C-elements, counters, and mutex elements).

We were interested in the detection of hazards in the wrapper. A hazard is a situation where, due to concurrently arriving input signals, it is not clear whether the corresponding output signal is able to fully change its value (after arrival of the first signal) before switching back to the original value. Such an incomplete signal switch may cause undefined signal values which are then potentially propagated through the whole circuit. The Petri net model was built such that the occurrence of a hazard in a particular gate corresponds to a particular reachable marking in the Petri net model of the gate.

LoLA was applied for solving the various reachability queries. Again, a combination of partial order reduction with the sweep-line method turned out to be most successful. Nevertheless, LoLA was not able to solve all of the reachability queries on the original model (having 286 places and 466 transitions). In a second approach, the model was replaced by a series of models each modeling one of the parts of the wrapper in detail while containing abstract versions of the others. The abstraction was based on the assumption that no hazards occur in the abstracted part. LoLA was then able to solve all reachability queries. The largest state space had little more than 10.000 nodes.

We detected eight hazards and reported them to the people in Frankfurt. For each hazard, we could derive a scenario for its occurrence from the witness paths available in LoLA. Six of the scenarios could be ruled out through knowledge about timing constraints. The remaining two hazards were considered as really dangerous situations. Using LoLA again, a re-design of the wrapper was verified as being hazard-free.

The approach of modeling one part of the investigated system in detail while abstracting the others was the main lesson learned out of this application

Garavel's Challenge

Back in 2003, Hubert Garavel posted a challenge to the Petri Net Mailing List. The mail contained a place/transition net with 485 places and 776 transitions that allegedly stemmed from a LOTOS specification. Garavel was interested in quasi-liveness of all transitions of the net.

Apart from LoLA, the two symbolic state space tools SMART (by G. Chiardo and R. Siminiceanu) and Versify (by O. Roig), and the tool TINA (by B. Berthomieu and F. Vernadat) which used the covering step graph technique responded to the challenge. The symbolic tools were able to calculate the exact number of reachable states of the system which was in the area of 10^{22}.

The LoLA approach to the challenge was to generate not just one (reduced) state space but 776 of them, one for the verification of quasi-liveness of a particular transition. This way, partial order reduction could be applied very successfully. 774 of the queries could be solved this way while two queries ran out of memory. For these transitions, we applied then the LoLA feature of generating random firing sequences. In fact, the sequences are not totally random. Instead, the probability of selecting a transition for firing is weighted according to a heuristics which is closely related to the stubborn set method. This heuristics is quite successful in attracting a firing sequence towards a state satisfying the investigated property. At least, it worked for the two problematic transitions in the challenge and we were able to report, for each transition, a path witnessing its quasi-liveness.

This challenge showed that LoLA can be competitive even to symbolic state space tools. A major reason for success was the division of the original verification problem into a large number of simpler verification tasks.

Exploring Biochemical Networks [Tal07]
In a biochemical network, a place represents a substance, tokens in a place represent presence of the substance. A transition models a known chemical reaction. A transition sequence that finally marks a place represents a chain of possible reactions that potentially generates the corresponding substance.

People at SRI use LoLA for exploring reaction paths. According to [Tal07], they “use LoLA because it is very fast in finding paths”.

The examples show that LoLA can be applied in various areas. It is able to cope with models of practically relevant systems. The performance of LoLA is due to at least four reasons:

– LoLA features a broad range of state-of-the-art state space reduction techniques most of which can be applied in combination;
– LoLA offers specialized reduction techniques for every property listed in the previous section;
– LoLA uses the formalism of place/transition nets which can be handled much easier than a high-level net formalism;
– The core procedures in LoLA exploit the basic characteristics of Petri nets as mentioned in the introduction.

4 Core Procedures in a State Space Generator

In this section, we demonstrate how the basic characteristics of Petri nets can be taken care of in the implementation of a state space generator. A state space generator is basically an implementation of a search through the state graph, typically a depth-first search. The elementary steps of a depth-first search include the following steps, each discussed in a dedicated subsection. When discussing complexity, we assume the investigated system to be *distributed*. Formally, we assume that there is a fixed value k such that every transition has, independently of the size of the net, at most k pre- and post-places. Several realistic distributed systems satisfy such a requirement for a reasonably small k, for even more systems there are only few transitions violating the assumption.

Firing a Transition

By firing a transition, we proceed from one state to a successor state. In a Petri net, the occurrence of a transition t changes the marking of at most $card(\bullet t) + card(t\bullet)$ places. According to the assumption made above, this number is smaller than $2k$. By maintaining, for each transition, an array of pre- and post-places, it is indeed possible to implement the occurrence of a transition in time $O(1)$. The ability to easily implement the firing process in constant time can be contributed to *locality*. In fact, other formalisms exhibiting locality have the same opportunity (like the model checking tool SPIN [Hol91] using the guarded command style language PROMELA. In contrast, the input language of the model checker SMV [McM02] does not support explicitly a notation of locality, and it would require a lot of expensive analysis for an explicit model checker to retrieve information on locality from SMV input. Note that SMV is not an explicit model checker, so this consideration does not concern SMV as such.

Checking Enabledness

The enabling status of a transition can change only due to the occurrence of a transition t. So, except, for an initial enabledness check on the initial marking, the check for enabledness can be reduced to the transitions in $\bullet t \cup t\bullet$. This approach is again feasible for all formalisms exhibiting *locality*. For Petri nets, however, the check for enabledness can be further refined due to the *monotonicity* of the enabling condition. If t' is enabled before having fired t, and t is only adding tokens to places in $\bullet t'$, it is clear that t' is still enabled after having fired t. Likewise, a previously disabled t' remains disabled if t only removes tokens from $\bullet t'$. This way, the number of enabledness checks after a transition occurrence can be significantly reduced. In LoLA, we maintain two separate lists of transitions for each t: those that can potentially be *enabled* by t (must be checked if they have been disabled before), and those that can be potentially disabled by t (must be checked if they have been enabled before). Through an additional treatment of all *enabled* transitions as a doubly linked list (with the opportunity to delete and insert an element at any position), it is possible to retrieve a list of enabled transitions in a time linear to the number of enabled transitions (which is typically an order of magnitude smaller than the overall number of transitions).

Returning to a Previously Seen Marking

In depth-first search, we typically have a *stack* of visited but not fully explored markings. This stack actually forms a path in the state space, that is, the immediate successor of marking m on the stack is reachable from m through firing a single transition. After having fully explored marking m on top of the stack, we proceed with its immediate predecessor m' on this stack. As Petri nets enjoy the *linearity* of the firing rule, there is a strikingly simple solution to this task: just fire the transition *backwards* that transformed m' to m. This way, it takes constant effort to get back to m'.

For assessing the value of this implementation, let us discuss potential alternatives. Of course, it would be possible to maintain a stack that holds full markings. Then returning to a previous marking amounts to redirecting a single pointer. But in depth-first exploration, the search stack typically holds a substantial number of visited states, so this approach would pay space for time. In state space verification, space is, however, the by far more limited resource. Another solution suggests to maintain, for each stack element, a pointer into the repository of visited markings. This data structure is, in principle, maintained in any explicit state space verification, so this solution would not waste memory at the first glance. For saving memory, it is, however, highly recommendable to deposit visited markings in a compressed form [WL93]. Thus, calculations on a marking in this compressed form require nontrivial run time. Finally, this solution prohibits approaches of storing only some reachable markings in the repository (see Sec. 5 for a discussion on such a method).

Maintaining the Visited Markings

According to the proposal to organize backtracking in the search by through firing transitions backwards, there are only two operations which need to be performed for on the set of visited markings. One operation is to search whether a newly encountered marking has been seen before, the other is to insert that marking if it has not. All other operations, including the evaluation of state predicates, computing the enabled transitions, computing successor and predecessor markings etc. can be performed on a single uncompressed variable, call it `CurrentMarking` (in the case of LoLA: an array of integers). For searching `CurrentMarking` and inserting it in the depository, we can look up and insert its compressed version.

In consequence, it is at no stage of the search necessary to uncompress a marking! This fact can be exploited for compressions where the uncompression is hard to realize. In LoLA, we have implemented such a technique [Sch03] that is based on place invariants (thus, a benefit from the *linearity* of the firing rule). Using a place invariant I, we can express the number of tokens of one place p in $supp(I)$ in terms of the others. We can thus exempt the value of p from being stored in any marking. Given n linearly independent place invariants, the number of values to be stored can be reduced by n. The number n typically ranges between 20% and 60% of the overall number of places, so the reduction is substantial. It does not only safe space but time as well. This is due to the fact that a look up in the depository is now performed on a smaller vector.

Compressing a marking according to this technique is rather easy: we just need to throw away places marked a "dependent" in a preprocessing stage. Uncompressing would require an evaluation of the justifying place invariant. In particular, it would be necessary to keep the invariant permanently in storage! In LoLA, we do not need to keep them. Concerning space, the additional costs of the approach, beyond preprocessing, amount to one bit ("dependent") for each place. Even in preprocessing, it is not necessary to fully compute the invariants. As explained in [Sch03], the information about mutual dependency can be deduced from an upper triangle form of the net incidence matrix, an intermediate stage of the calculation. This, way, invariant based preprocessing requires less than a second of run time even for a net with 5.000 places and 4.000 transitions. In that particular system, we would have 2.000 linearly independent place invariants (each being a vector of length 5.000!).

Breadth-First Search
While depth-first search is the dominating technique for state space exploration, breadth-first search can be used for some purposes as well, for instance for the calculation of a shortest path to some state. In breadth-first search, subsequently considered states are not connected by a transition occurrence. Nevertheless, it is possible to preserve some of the advantages of the backtracking through firing transitions. In LoLA, we mimic breadth-first search by a depth-first search with an incrementally increased depth restriction. That is, we proceed to the next marking to be considered by stepping back a few markings (through firing some transitions backwards) and then firing some other transitions forward. The average number of transitions to be fired is reasonably small as the number of states tends to grow exponentially with increased depth. This is true even for reduced state spaces, as some of the most powerful reduction techniques require the use of depth-first search.

5 Reduction Techniques

In this section, we discuss a few state space reduction techniques and show that the basic characteristics of Petri nets lead to specific solutions.

Partial Order Reduction
Roughly spoken, the purpose of partial order reduction is to suppress as many as possible interleaved firings of concurrently enabled transitions. This goal is achieved by considering, in each marking, only a subset of the enabled transitions. This subset is computed such that a given property or class of properties is preserved in the reduced state space.

It is well-known that *locality* is the major pre-requisite of the stubborn set method [Val88] and other methods of partial order reduction [Pel93, GW91, GKPP95]. Furthermore, *linearity* of the firing rule turns out to be quite beneficial. The reason is that partial order reduction is, among others, concerned with permutations of firing sequences. It is typically desired that a firing sequence reaches the same marking as the permuted sequence. Due to the *linearity* of the firing rule, this property comes free for Petri nets. For other formalisms, it needs to be enforced, as can be seen in [Val91]. This way, other formalisms have additional limitations in the application of partial order reduction.

For partial order reduction, there is another source of efficiency that is worth being mentioned. It is not related to the formalism of Petri nets itself, but with the tradition of Petri net research. In the area of Petri nets, people have studied a broad range of singular properties such as boundedness, liveness, reversibility, reachability, deadlock freedom, etc. Following this tradition, it was apparent to support each of these properties with a *dedicated* version of partial order reduction [Sch99,Sch00d,KV00,KSV06]. In contrast, it is the tradition of model checking to support a rich language or two (such as the temporal logics CTL [Eme90] or LTL [MP92]). According this line of research, people came up with reduction techniques that support the whole language [Pel93, GKPP95]. It is evident, that a dedicated reduction technique for property X can lead to a better reduction than a generic technique for a specification language can can express X. We believe that this observation is crucial for the competitivity of LoLA in various areas.

The Symmetry Method

Symmetrically structured systems exhibit a symmetric behavior. Exploiting symmetry means to suppress consideration of a state if a symmetric state has been considered before.

Most approaches search for symmetric structures in data types of the specification. The most popular data type in this respect is the so-called *scaler set* [DDHC92] where variables can be compared for equality, used as indices in arrays and order-independent loops, while there are no constants of that type. In [CDFH90], a rather sophisticated detection of symmetric structure in data types is described.

Due to the *locality* of Petri nets, place/transition nets have a rather fine grained graphical representation. This feature enables another approach to finding symmetries in the structure: we can compute the graph automorphisms of the Petri net graph [Sta91, Sch00a, Sch00b, Jun04]. There is a polynomial size generating set of the automorphism group of a graph, and it can be computed in reasonable time (though not always in polynomial time). The generating set is sufficient for an approximated calculation of a canonical representative of a marking [Sch00b], a method for detecting previously seen symmetric states during state space calculation. The graph automorphism based approach to symmetry is a unique feature of LoLA and INA [RS98] (both implemented by the author of this article).

The main advantage of the graph automorphism approach is that it can recognize *arbitrary* symmetry groups while the data type approach is restricted to a couple of standard symmetries.

The Sweep-Line Method

The sweep-line method assumes that there is a notion of *progress* in the system evolution. That is, assigning a progress value to each state, successor markings tend to have larger progress values than their predecessors. This observation can be exploited by traversing the search space in order of increasing progress values, and to remove visited markings from the depository which have smaller progress value than the currently considered markings. For reason of correctness, markings which are reached through a transition that decreases the progress value, are stored permanently, and their successors are encountered.

The original method [Mai03,CKM01,KM02] assumes that the progress measure is given manually. However, exploiting the *linearity* of the Petri net firing rule, it is possible to compute a progress measure automatically. The measure being calculated assigns some arbitrary progress value, say 0, to the initial state. Then, each transition t gets an offset $o(t)$ such that, if t fired in m leads to m', the progress value of m' is just the progress value of m plus $o(t)$. For correctness, it is important that different firing sequences from the initial marking to a marking m all yield the same progress value for m. This can, however, been taken care of by studying linear dependencies between transition vectors. In LoLA, we compute the measure by assigning an arbitrary offset, say 1, to each transition in a maximum size linearly independent set U of transitions. For the remaining transitions (which are linear combinations of U) the offset is then determined by the correctness requirement stated above. All applications of the sweep-line method reported in this article have been conducted with an automatically computed progress measure.

Cycle Coverage

The depository of visited markings is the crucial date structure in a state space verification. In explicit methods, the size of the depository grows with the number of visited states. The number of states to be stored can, however, be reduced in a trade that sells time for space. By simply exempting states from being stored, we obviously safe space but lose time as, in a revisit to a forgotten state, its successors are computed once again. For an implementation of this idea, it is, as for instance observed in [LLPY97], important to store at least one marking of each cycle in the state graph. This condition actually ensure termination of the approach.

Thanks to *linearity* in the Petri net firing rule, it is fairly easy to characterize a set of states such that every cycle in the state graph is covered. We know that every firing sequence that reproduces the start marking forms a transition invariant. Thus, choosing a set of transitions U such that the support of every transition invariant contains an element from U, it is evident that every cycle in the state graph contains at least one marking where a transition in U is enabled. This approach has been described in [Sch03].

Combination of Reduction Techniques

Most techniques mentioned above can be applied in combination. The combined application typically leads to additional reduction like in the case of joint application of partial order reduction with the symmetry method. For some reduction techniques, we experienced that their joint application with another technique is actually a pre-requisite for a substantial reduction as such. For instance, the sweep-line method only leads to marginal reduction for Petri nets with a lot of cycles [Sch04]. Also, the cycle coverage reduction does not perform well on such systems [Sch03]. Both methods can, however, lead to substantial (additional!) reduction when they are applied to a stubborn set reduced state space. This is due to a particular effect of partial order reduction. If a system consists of several, mostly concurrently evolving, cyclic components, then the almost arbitrary interleaving of transitions in these components closes a cycle in almost every reachable state. This causes a tremendous number of regress transitions in the sweep-line method (and thus a huge number of states to be stored permanently) and a huge number of states to be stored with the cycle coverage reduction. Partial order reduction decouples the arbitrary interleaving of concurrent components. A partial order reduced state space contains only a fraction of the cycles of the original state space, and the remaining cycles tend to be significantly larger.

6 Conclusion

Petri nets as a formalism for modeling systems enjoy specific properties including locality, linearity and monotonicity. These properties lead to specific verification techniques such as the coverability graph, the invariant calculus, siphon/trap based analyses, or the unfolding approach. In this article we demonstrated, that the specific properties of Petri nets are as well beneficial for the implementation of techniques which are otherwise applicable in other formalisms as well. Our discussion covered explicit state space verification as such, but also a number of state space reduction techniques all of which can be applied to several modeling languages.

All mentioned methods have been implemented in the tool LoLA. LoLA is able to solve problems that have practical relevance. We hold three reasons responsible for the performance of LoLA:

– A consistent exploitation of the basis characteristics of Petri nets,
– A broad variety of reduction techniques which can be applied in many combinations, and
– The availability of dedicated reduction techniques for frequently used singular properties.

In this light, it is fair to say that LoLA is a *Petri net* state space tool.

References

[Ber86] Berthelot, G.: Checking properties of nets using transformations. Advances in Petri Nets, pp. 19–40 (1986)

[CDFH90] Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well–formed colored nets and their symbolic reachability graph. In: Proc. ICATPN, pp. 378–410 (1990)

[CE81] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronisation skeletons using brnaching tim temporal logic. In: Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)

[CKM01] Christensen, S., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)

[Com72] Commoner, F.: Deadlocks in Petri nets. Technical report, Applied Data Research Inc. Wakefield, Massachussetts (1972)

[Cur+03] Curbera, F., et al.: Business process execution language for web services, version 1.1. Technical report, BEA, IBM, Microsoft (2003)

[DDHC92] Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: Proc. IEEE Int. Conf. Computer Design: VLSI in Computers and Processors, pp. 522–525 (1992)

[Eme90] Emerson, E.A.: Handbook of Theoretical Computer Science. Chapter 16. Elsevier, Amsterdam (1990)

[Esp92] Esparza, J.: Model checking using net unfoldings. Technical Report 14/92, Universität Hildesheim (1992)

[Fin90] Finkel, A.: A minimal coverability graph for Petri nets. In: Proc. ICATPN, pp. 1–21 (1990)

[GKPP95] Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. In: Symp. on the Theory of Computing and Systems, IEEE, pp. 130–140 (1995)

[GL83] Genrich, H., Lautenbach, K.: S–invariance in Pr/T–nets. Informatik–Fachberichte 66, 98–111 (1983)

[GW91] Godefroid, P., Wolper, P.: A partial approach to model checking. In: IEEE Symp. on Logic in Computer Science, pp. 406–415 (1991)

[Hol91] Holzmann, G.: Design an Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs (1991)

[HSS05] Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: Proc. BPM, LNCS 3649, pp. 220–235 (2005)

[Jen81] Jensen, K.: Coloured Petri nets and the invariant method. Theoretical Computer Science 14, 317–336 (1981)

[Jun04] Junttila, T.: New canonical representative marking algorithms for place/transition nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 258–277. Springer, Heidelberg (2004)

[KM69] Karp, R.M., Miller, R.E.: Parallel programm schemata. Journ. Computer and System Sciences 4, 147–195 (1969)

[KM02] Kristensen, L.M., Mailund, T.: A generalized 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)

[KPA99] Kordon, F., Paviot-Adet, E.: Using CPN-AMI to validate a sfae channel protocol. Tool presentation at ICATPN (1999)

[KSV06] Kristensen, L., Schmidt, K., Valmari, A.: Question-guided stubborn set methods for state properties. Formal Methods in System Design 29(3), 215–251 (2006)

[KV00] Krisensen, L.M., Valmari, A.: Improved question-guided stubborn set methods for state properties. In: Proc. ICATPN, pp. 282–302 (2000)

[KW01] Kindler, E., Weber, M.: The Petri net kernel - an infrastructure for building petri net tools. STTT 3 (4), 486–497 (2001)

[LLPY97] Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: Proc. IEEE Real-Time Systems Symp., pp. 14–24 (1997)

[LS74] Lautenbach, K., Schmidt, H.A.: Use of Petri nets for proving correctness of concurrent process systems. IFIP Congress, pp. 187-191 (1974)

[McM02] McMillan, K.: The SMV homepage. `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`

[Mai03] Mailund, T.: Sweeping the State Space - a sweep-line state space exploration method. PhD thesis, University of Aarhus (2003)

[MP92] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems, vol. 1: Specification. Springer, Heidelberg (1992)
[McM92] McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronious circuits. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993)
[Pel93] Peled, D.: All from one, one for all: on model–checking using representitives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
[QS81] Quielle, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1981)
[RS98] Roch, S., Starke, P.: INA Integrierter Netz Analysator Version 2.1. Technical Report 102, Humboldt–Universität zu Berlin (1998)
[Sch99] Schmidt, K.: Stubborn set for standard properties. In: Donatelli, S., Kleijn, J.H.C.M. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 46–65. Springer, Heidelberg (1999)
[Sch00a] Schmidt, K.: How to calculate symmetries of Petri nets. Acta Informatica 36, 545–590 (2000)
[Sch00b] Schmidt, K.: Integrating low level symmetries into reachability analysis. In: Schwartzbach, M.I., Graf, S. (eds.) ETAPS 2000 and TACAS 2000. LNCS, vol. 1785, pp. 315–331. Springer, Heidelberg (2000)
[Sch00c] Schmidt, K.: LoLA – a low level analyzer. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000)
[Sch00d] Schmidt, K.: Stubborn set for modelchecking the EF/AG fragment of CTL. Fundamenta Informaticae 43 (1-4), 331–341 (2000)
[Sch03] Schmidt, K.: Using Petri net invariants in state space construction. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 473–488. Springer, Heidelberg (2003)
[Sch04] Schmidt, K.: Automated generation of a progress measure for the sweep-line method. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 192–204. Springer, Heidelberg (2004)
[SSE03] Schröter, C., Schwoon, S., Esparza, J.: The Model-Checking Kit. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 463–472. Springer, Heidelberg (2003)
[SRK05] Stahl, C., Reisig, W., Krstic, M.: Hazard detection in a GALS wrapper: A case study. In: Proc. ACSD, pp. 234–243 (2005)
[Sta91] Starke, P.: Reachability analysis of Petri nets using symmetries. J. Syst. Anal. Model. Simul. 8, 294–303 (1991)
[Tal07] Talcott, C.: Personal communication. Dagstuhl Seminar (February 2007)
[T4B07] Reisig, W., et al.: The homepage of the project Tools4BPEL `http://www2.informatik.hu-berlin.de/top/forschung/projekte/tools4bpel/`
[Val88] Valmari, A.: Error detection by reduced reachability graph generation. In: ICATPN (1988)
[Val91] Valmari, A.: Stubborn sets for reduced state space generation. In: Advances in Petri Nets 1990. LNCS, vol. 483, pp. 491–511. Springer, Heidelberg (1991)
[WL93] Wolper, P., Leroy, D.: Reliable hashing without collision detection. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)

Markov Decision Petri Net and Markov Decision Well-Formed Net Formalisms

M. Beccuti, G. Franceschinis[1,⋆], and S. Haddad[2,⋆⋆]

[1] Univ. del Piemonte Orientale
`giuliana.franceschinis@mfn.unipmn.it`
[2] LAMSADE CNRS, Univ. Paris Dauphine
`haddad@lamsade.dauphine.fr`

Abstract. In this work, we propose two high-level formalisms, *Markov Decision Petri Nets* (MDPNs) and *Markov Decision Well-formed Nets* (MDWNs), useful for the modeling and analysis of distributed systems with probabilistic and non deterministic features: these formalisms allow a high level representation of Markov Decision Processes. The main advantages of both formalisms are: a macroscopic point of view of the alternation between the probabilistic and the non deterministic behaviour of the system and a syntactical way to define the switch between the two behaviours. Furthermore, MDWNs enable the modeller to specify in a concise way similar components. We have also adapted the technique of the symbolic reachability graph, originally designed for Well-formed Nets, producing a reduced Markov decision process w.r.t. the original one, on which the analysis may be performed more efficiently. Our new formalisms and analysis methods are already implemented and partially integrated in the GreatSPN tool, so we also describe some experimental results.

1 Introduction

Markov Decision Processes (MDP). Since their introduction in the 50's, Markov Decision process models have gained recognition in numerous fields including computer science and telecommunications [13]. Their interest relies on two complementary features. On the one hand, they provide to the modeler a simple mathematical model in order to express optimization problems in random environments. On the other hand, a rich theory has been developed leading to efficient algorithms for most of the practical problems.

Distributed Systems and MDPs. The analysis of distributed systems mainly consists in (1) a modeling phase with some high-level formalism like Petri nets (PN) or process algebra, (2) the verification of properties expressed in some logic (like LTL or CTL) and (3) the computation of performance indices by enlarging the model with stochastic features and applying either (exact or approximate)

⋆ The work of these authors was supported in part with the Italian MUR "local" research funds.

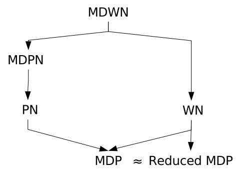
⋆⋆ The work of this author was supported in part by ANR Setin project CheckBound.

analysis methods or simulations. In this framework, a MDP may be viewed as a model of a distributed system where it is possible to perform a non deterministic choice among the enabled actions (*e.g.*, the scheduling of tasks) while the effect of the selected action is probabilistic (*e.g.*, the random duration of a task). Then, with appropriate techniques, one computes the probability that a property is satisfied w.r.t. the "worst" or the "best" behavior [3,7]. The time model that will be considered in this paper is discrete: each non deterministic choice is taken in a given decision epoch, after the probabilistic consequence of the choice has been performed, a new decision epoch starts.

Here the way we model distributed systems by MDPs is rather different. During a phase, the system evolves in a probabilistic manner until periodically a (human or automatic) supervisor takes the control in order to configure, adapt or repair the system depending on its current state before the next phase. In other words, usual approaches consider that the alternation between non deterministic and probabilistic behavior occurs at a microscopic view (*i.e.*, at the transition level) whereas our approach adopts a macroscopic view of this alternation (*i.e.*, at a phase level). It should be emphasized that, depending on the applications, one or the other point of view should be preferred and that the user should have an appropriate formalism and associated tools for both cases. For instance PRISM [11], one of the most used tools in this context, works at the microscopic level whereas the formalism of *stochastic transition systems* is based on a macroscopic view [8]. The latter formalism is a slight semantical variation of generalized stochastic Petri nets [12] where the choice among the enabled immediate transitions is performed non deterministically rather than probabilistically. Despite its simplicity, this formalism has a serious drawback for the design process since the modeler has no mean to syntactically define the switches between the probabilistic behavior and the non deterministic one. Furthermore, the difference between the *distributed* feature of the probabilistic behavior and the *centralized* one of the non deterministic behavior is not taken into account.

Our contribution. In this work, we propose a high-level formalism in order to model distributed systems with non deterministic and probabilistic features. Our formalism is based on *Well-formed Petri Nets* (WN) [4]. First, we introduce *Markov Decision Petri nets* (MDPN): an MDPN is defined by three parts, a set of active components (*e.g.*, processes or machines), a probabilistic net and a non deterministic net. Every transition of the probabilistic net is triggered by a subset of components. When every component has achieved the activities related to the current probabilistic phase, the supervisor triggers the non deterministic transitions in order to take some decisions, either relative to a component or global. Every transition has an attribute (run/stop) which enables the modeler to define when the switches between the nets happen. The semantics of this model is designed in two steps: a single Petri net can be derived from the specification and its reachability graph can be transformed with some additional information, also specified at the MDPN level, into an MDP.

Distributed systems often present symmetries *i.e*, in our framework, many components may have a similar behavior. Thus, both from a modeling and an analysis point of view, it is interesting to look for a formalism expressing and exploiting behavioral symmetries. So we also define *Markov Decision Well-formed nets* (MDWN) similarly as we do for MDPNs. The semantics of a model is then easily obtained by translating a MDWN into a MDPN. Furthermore, we develop an alternative approach: we transform the MDWN into a WN, then we build the symbolic reachability graph of this net [5] and finally we transform this graph into a reduced MDP w.r.t. the original one. We argue that we can compute on this reduced MDP, the results that we are looking for in the original MDP. The different relations between the formalisms are shown in the figure depicted below. Finally we have implemented our analysis method within the GreatSPN tool [6] and performed some experiments.



Organization of the paper. In section 2, we recall basic notions relative to MDPs, then we define and illustrate MDPNs. In section 3, we introduce MDWNs and develop the corresponding theoretical results. In section 4, we present some experimental results. In section 5, we discuss related work. Finally we conclude and give some perspectives in section 6.

2 Markov Decision Petri Net

2.1 Markov Decision Process

A (discrete time and finite) MDP is a dynamic system where the transition between states (*i.e.*, items of S a finite set) are obtained as follows. First, given s the current state, one non deterministically selects an action among the subset of actions currently enabled (*i.e.*, A_s). Then one samples the new state w.r.t. to a probability distribution depending on s and $a \in A_s$ (*i.e.*, $p(\cdot|s,a)$). An MDP includes rewards associated with state transitions; here, we choose a slightly restricted version of the rewards that do not depend on the destination state (*i.e.*, $r(s,a)$). Starting from such elementary rewards, different kinds of global rewards may be associated with a finite or infinite execution thus raising the problem to find an optimal strategy w.r.t. a global reward. For sake of simplicity, we restrict the global rewards to be either the *expected total reward* or the *average reward.* The next definitions formalize these concepts.

Definition 1 (MDP). *An MDP $\mathcal{M}$ is a tuple $\mathcal{M} = \langle S, A, p, r\rangle$ where:*

- *S is a finite set of states,*
- *A is a finite set of actions defined as $\bigcup_{s\in S} A_s$ where A_s is the set of enabled actions in state s,*

– $\forall s \in S, \forall a \in A_s, p(\cdot|s,a)$ *is a (transition) probability distribution over* S *such that* $p(s'|s,a)$ *is the probability to reach* s' *from* s *by triggering action* a,
– $\forall s \in S, \forall a \in A_s, r(s,a) \in \mathbb{R}$ *is the reward associated with state* s *and action* a.

A finite (resp. infinite) execution of an MDP is a finite (resp. infinite) sequence $\sigma = s_0a_0 \dots s_n$ *(resp.* $\sigma = s_0a_0 \dots$*) of alternating states and actions, s.t.* $\forall i, s_i \in S \wedge a_i \in A_{s_i}$ *and* $p(s_{i+1}|s_i,a_i) > 0$.

The total reward of such an execution is defined by $trw(\sigma) = \sum_{i=0}^{n-1} r(s_i,a_i)$ *(resp.* $trw(\sigma) = \lim_{n\to\infty} \sum_{i=0}^{n-1} r(s_i,a_i)$ *provided the limit exists) and its average reward is* $arw(\sigma) = \frac{1}{n}\sum_{i=0}^{n-1} r(s_i,a_i)$ *(resp.* $arw(\sigma) = \lim_{n\to\infty} \frac{1}{n}\sum_{i=0}^{n-1} r(s_i,a_i)$ *provided the limit exists).*

We denote $SEQ^*(\mathcal{M})$ (resp. $SEQ^\infty(\mathcal{M})$) the set of finite (resp. infinite) sequences. A strategy st is a mapping from $SEQ^*(\mathcal{M})$ to A such that $st(s_0a_0 \dots s_n)$ belongs to A_{s_n}. Since a strategy discards non determinism, the behavior of $\mathcal{M}$ w.r.t. st is a stochastic process $\mathcal{M}^{st}$ defined as follows. Assume that the current execution is some $s_0a_0 \dots s_n$ then $a_n = st(s_0a_0 \dots s_n)$ and the next state s_{n+1} is randomly chosen w.r.t. distribution $p(\cdot|s_n,a_n)$. Consequently, the reward of a random sequence of $\mathcal{M}^{st}$ is a random variable and the main problem in the MDP framework is to maximize or minimize the mean of this random variable and to compute the associated strategy when it exists. In finite MDPs, efficient solution techniques have been developed to this purpose [13].

Here we want to model systems composed by multiple active components whose behavior during a period is described in a probabilistic way and a centralized decision maker taking some decisions between execution periods (*e.g.*, assigning available resources to components). Let us illustrate this kind of systems by a toy example. Imagine an information system based on two redundant computers: this system is available as long as one computer is in service. A computer may fail during a period. At the end of a period, the decision maker can choose to send a single repairman to repair a faulty computer when he is not yet busy. There is a fixed probability that the repairing ends inside the period. In this framework, the rewards denote costs (for unavailability and repairs) and the analysis aims at minimizing them. The MDP corresponding to this system is shown in Fig. 1, where the states description does not maintain the distinction between the components; this is only possible when the computers are identical.

The design of this MDP is rather easy. However when the system has more computers and repairmen with different behaviors, then modeling it at the MDP level becomes unfeasible.

2.2 Markov Decision Petri Net

A Markov Decision Petri Net $\mathcal{MN}$ is composed by two different parts (i.e. two extended Petri nets): the probabilistic one N^{pr} and the non deterministic one N^{nd} called the *decision maker*; it is thus possible to clearly distinguish and design the probabilistic behavior of the system and the non deterministic one. The probabilistic part models the probabilistic behavior of the system and can

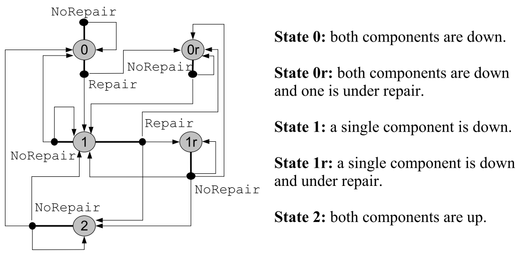

NoRepair
Repair
NoRepair
Repair
NoRepair
NoRepair
NoRepair
0
0r
1
1r
2
State 0: both components are down.
State 0r: both components are down and one is under repair.
State 1: a single component is down.
State 1r: a single component is down and under repair.
State 2: both components are up.

Fig. 1. Symbolic representation of the MDP modeling in this section

be seen as composition of a set of n components ($Comp^{pr}$) that can interact; instead the non deterministic part models the non deterministic behavior of the system where the decisions must be taken (we shall call this part *the decision maker*). Hence the global system behavior can be described as an alternating sequence of probabilistic and non deterministic phases.

The probabilistic behavior of a component is characterized by two different types of transitions $Trun^{pr}$ and $Tstop^{pr}$. The $Trun^{pr}$ transitions represent intermediate steps in a probabilistic behavior phase and can involve several components (synchronized through that transition), while the $Tstop^{pr}$ ones always represent the final step of the probabilistic phase of at least one component.

In the non deterministic part, the decisions can be defined at the system level (transitions of T_g^{nd}) or at the component level (transitions of T_l^{nd}). The sets T_g^{nd} and T_l^{nd} are again partitioned in $Trun_g^{nd}$ and $Tstop_g^{nd}$, and $Trun_l^{nd}$ and $Tstop_l^{nd}$ with the same meaning. The decision maker does not necessarily control every component and may not take global decisions. Thus the set of controllable "components" $Comp^{nd}$ is a subset of $Comp^{pr} \uplus \{id_s\}$ where id_s denotes the whole system.

The probabilistic net is enlarged with a mapping *weight* associating a weight with every transition in order to compute the probabilistic choice between transitions enabled in a marking. Furthermore it includes a mapping *act* which associates to every transition the subset of components that (synchronously) trigger the transition. The non deterministic net is enlarged with a mapping *obj* which associates with every transition the component which is involved by the transition. The following definition summarizes and formalizes this presentation.

Definition 2 (Markov Decision Petri Net (MDPN)). *A Markov Decision Petri Net (MDPN) is a tuple* $\mathcal{MN} = \langle Comp^{pr}, Comp^{nd}, N^{pr}, N^{nd}\rangle$ *where:*

– $Comp^{pr}$ *is a finite non empty set of components;*
– $Comp^{nd} \subseteq Comp^{pr} \uplus \{id_s\}$ *is the non empty set of controllable components;*
– N^{pr} *is defined by a PN with priorities [12]* $\langle P, T^{pr}, I^{pr}, O^{pr}, H^{pr}, prio^{pr}, m_0\rangle$, *a mapping weight:* $T^{pr} \to \mathbb{R}$ *and a mapping act:* $T^{pr} \to 2^{Comp^{pr}}$. *Moreover* $T^{pr} = Trun^{pr} \uplus Tstop^{pr}$
– N^{nd} *is defined by a PN with priorities* $\langle P, T^{nd}, I^{nd}, O^{nd}, H^{nd}, prio^{nd}, m_0\rangle$ *and a mapping obj:* $T^{nd} \to Comp^{nd}$. *Moreover* $T^{nd} = Trun^{nd} \uplus Tstop^{nd}$.

Furthermore, the following constraints must be fulfilled:

- $T^{pr} \cap T^{nd} = \emptyset$. *A transition cannot be non deterministic and probabilistic.*
- $\forall id \in Comp^{pr}, \exists C \subseteq Comp^{pr}$, *s.t.* $id \in C$ *and* $act^{-1}(\{C\}) \cap Tstop^{pr} \neq \emptyset$. *Every component must trigger at least one* final *probabilistic transition.*
- $\forall id \in Comp^{nd}, obj^{-1}(\{id\}) \cap Tstop^{nd} \neq \emptyset$. *Every controllable component must be the object of at least one* final *non deterministic transition.*

Note that the probabilistic part and the decision maker share the same set of places and the same initial marking. Let us now introduce the rewards associated with the MDPN net. As will be developed later, an action of the decision maker corresponds to a sequence of transition firings starting from some marking. We choose to specify a reward by first associating with every marking m a reward $rs(m)$, with every transition t a reward $rt(t)$ and then by combining them with an additional function rg (whose first parameter is a state reward and the second one is a reward associated with a sequence of transition firings). The requirement on its behavior given in the next definition will be explained when presenting the semantics of a MDPN.

Definition 3 (MDPN reward functions). *Let $\mathcal{MN}$ be a MDPN. Then its reward specification is given by:*

- $rs : \mathbb{N}^P \to \mathbb{R}$ *which defines for every marking its reward value.*
- $rt : T^{nd} \to \mathbb{R}$ *which defines for every transition its reward value.*
- $rg : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, *not decreasing w.r.t its second parameter.*

An example of (a portion of) probabilistic and non deterministic subnets is shown in Fig. 3: in the framework of the MDPN formalism, the annotations on arcs and next to the places and transitions should be ignored. The decision maker implements the possible ways of assigning resources (e.g. for component repair) to those components that need them (e.g. failed components): for each component needing a resource two possibilities are included, namely assign or not assign resource. The probabilistic part shows 3 system components: two of them are controllable (let's call them Proc and Mem), one is not controllable (let's call it ResCtr). The Proc and Mem components can work fine or fail, the third one supervises the repair process (when the resource is available) and the Proc and/or Mem resume phase. $Tstop$ transitions are e.g. *WorkFineProc*, *WaitRepProc*, *ResumeProc*, *ResumeMemProc*, the first two involving only Proc, the third involving Proc and ResCtr, the last one involving all three components; the firing of these transitions mean that the involved components have reached a stable state in the current decision epoch. $Trun$ transitions are e.g. *FailProc*, *FailMem* involving respectively Proc and Mem, and *EndRep* involving only ResCtr. These transitions represent intermediate steps in the components evolution (e.g. a failure can be followed by a wait for repair resource or a resume final step).

MDPN Semantics. The MDPN semantics is given in three steps. First, one composes the probabilistic part and the decision maker in order to derive a unique PN. Then one generates the (finite) reachability graph (RG) of the PN. At last, one produces an MDP from it.

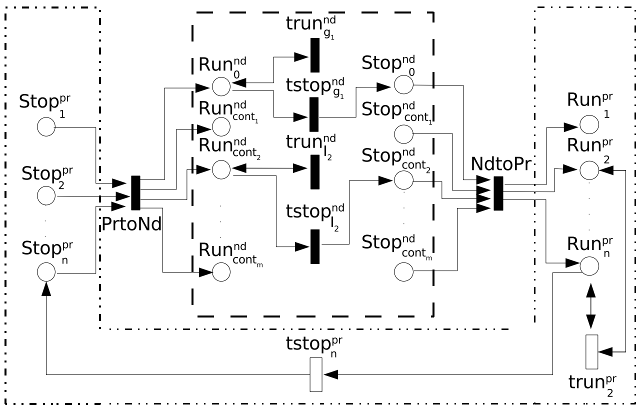


Fig. 2. Arcs connecting the places $Stop_i^{pr}$, Run_i^{nd} and the transition $PrtoNd$; arcs connecting the places $Stop_i^{nd}$, Run_i^{nd} and the transition $NdtoPr$

From MDPN to PN. First we explain the semantics of additional places $Stop_i^{pr}$, Run_i^{pr}, $Stop_i^{nd}$, Run_i^{nd}, $Stop_0^{nd}$ and $Run_0^{nd}$and additional non deterministic transitions $PrtoNd$ and $NdtoPr$. Places $Stop_i^{pr}$, Run_i^{pr}, $Stop_i^{nd}$, Run_i^{nd}, $Stop_0^{nd}$ and Run_0^{nd} regulate the interaction among the components, the global system and the decision maker. There are places Run_i^{pr}, $Stop_i^{pr}$ for every component i, while we insert the places Run_0^{nd} and $Stop_0^{nd}$ if the decision maker takes same global decision and the pair of places Run_i^{nd} and $Stop_i^{nd}$ for every controllable component $i \in Comp^{nd}$. Non deterministic transitions $PrtoNd$ and $NdtoPr$ ensure that the decision maker takes a decision for every component in every time unit: the former triggers a non deterministic phase when all the components have finished their probabilistic phase whereas the latter triggers a probabilistic phase when the decision maker has taken final decisions for every controllable component.

The scheme describing how these additional items are connected together and with the nets of the MDPN is shown in Fig. 2. The whole PN $N^{comp} = \langle P^{comp}, T^{comp}, I^{comp}, O^{comp}, H^{comp}, prio^{comp}, m_0^{comp}\rangle$ related to a MDPN $\mathcal{MN}$ is defined below.

- $P^{comp} = P \uplus_{i\in Comp_{pr}} \{Run_i^{pr}, Stop_i^{pr}\} \uplus_{i\in Comp_{nd}} \{Run_i^{nd}, Stop_i^{nd}\}$
- $T^{comp} = T^{pr} \uplus T^{nd} \uplus \{PrtoNd, NdtoPr\}$
- The incidence matrices of N^{comp} are defined by:
  - $\forall p \in P, t \in T^{nd}$,
    $I^{comp}(p,t) = I^{nd}(p,t), O^{comp}(p,t) = O^{nd}(p,t), H^{comp}(p,t) = H^{nd}(p,t)$
  - $\forall p \in P, t \in T^{pr}$,
    $I^{comp}(p,t) = I^{pr}(p,t), O^{comp}(p,t) = O^{pr}(p,t), H^{comp}(p,t) = H^{pr}(p,t)$
  - $\forall t \in Tstop^{pr}$ *s.t.* $i \in act(t) : I^{comp}(Run_i^{pr}, t) = O^{comp}(Stop_i^{pr}, t) = 1$
  - $\forall t \in Trun^{pr}$ *s.t.* $i \in act(t) : I^{comp}(Run_i^{pr}, t) = O^{comp}(Run_i^{pr}, t) = 1$
  - $\forall t \in Tstop^{nd}$ *s.t.* $i \in act(t) : I^{comp}(Run_i^{nd}, t) = O^{comp}(Stop_i^{nd}, t) = 1$
  - $\forall t \in Trun^{nd}$ *s.t.* $i \in act(t) : I^{comp}(Run_i^{nd}, t) = O^{comp}(Run_i^{nd}, t) = 1$

  - $\forall i \in Comp_{pr} : I^{comp}(Stop_i^{pr}, PrtoNd) = O^{comp}(Run_i^{pr}, NdtoPr) = 1$
  - $\forall i \in Comp_{nd} : I^{comp}(Stop_i^{nd}, NdtoPr,) = O^{comp}(Run_i^{nd}, PrtoNd) = 1$
  - for all $I(p,t), O(p,t), H(p,t)$ not previously defined, $I^{comp}(p,t) = O^{comp}(p,t) = 0, H^{comp}(p,t) = \infty$;
- $\forall t \in T^{nd}, prio(t) = prio^{nd}(t)$, $\forall t \in T^{pr}, prio(t) = prio^{pr}(t)$, $prio(PrtoNd) = prio(NdtoPr) = 1$, (actually these values are irrelevant)
- $\forall p \in P, m_0^{Comp}(p) = m_0(p)$, $m_0^{Comp}(Run_i^{nd}) = 1$, $m_0^{Comp}(Stop_i^{nd}) = m_0^{Comp}(Run_i^{pr}) = m_0^{Comp}(Stop_i^{pr}) = 0$.

RG semantics and transitions sequence reward. Considering the RG obtained from the PN we observe that the reachability set (RS) can be partitioned into two subsets: the non deterministic states (RS_{nd}), in which only non deterministic transitions are enabled, and the probabilistic states (RS_{pr}), in which only probabilistic transitions are enabled. By construction, the PN obtained from a MDPN can never reach a state enabling both nondeterministic and probabilistic transitions. A probabilistic transition can be enabled only if there is at least one place Run_i^{pr} with $m(Run_i^{pr}) > 0$, while a non deterministic transition can be enabled only if there is at least one place Run_i^{nd} with $m(Run_i^{nd}) > 0$. Initially only Run_i^{nd} places are marked. Then only when all the tokens in the Run_i^{nd} places have moved to the $Stop_i^{nd}$ places (through the firing of some transition in $Tstop^{nd}$), the transition $NdtoPr$ can fire, removing all tokens from the $Stop_i^{nd}$ places and putting one token in every Run_i^{pr} place. Similarly, transition $PrtoNd$ is enabled only when all tokens have moved from the Run_i^{pr} to the $Stop_i^{pr}$ places; the firing of $PrtoNd$ brings the tokens back in each Run_i^{nd} place. Thus places Run_i^{pr} and places Run_i^{nd} cannot be simultaneously marked.

Observe that any *path* in the RG can be partitioned into (maximal) sub-paths leaving only states of the same type, so that each path can be described as an alternating sequence of non deterministic and probabilistic sub-paths. Each probabilistic sub-path can be substituted by a single "complex" probabilistic step and assigned a probability based on the weights of the transitions firing along the path. The non deterministic sub-paths can be interpreted according to different semantics (see [2] for a detailed discussion). Here we select the following semantics: a path through non deterministic states is considered as a single complex action and the only state where time is spent is the first one in the sequence (that is the state that triggers the "complex" decision multi-step). So only the first state in each path will appear as a state in the MDP (the other states in the path are *vanishing*, borrowing the terminology from the literature on generalized stochastic Petri nets).

Let us now define the reward function for a sequence of non deterministic transitions, $\sigma \in (T^{nd})^*$; abusing notation we use the same name $rt()$ for the reward function for single transitions and for transition sequences. The following definition $rt(\sigma)$ assumes that the firing order in such a sequence is irrelevant w.r.t. the reward which is consistent with an additive interpretation when several decisions are taken in one step.

Definition 4 (Transition sequence reward $rt(\sigma)$). *The reward for a non deterministic transition sequence is defined as follows:*

$$rt(\sigma) = \sum_{t \in T^{nd}} rt(t)|\sigma|_t$$

where $|\sigma|_t$ is the number of occurrences of non deterministic transition t in σ.

Generation of an MDP given a RG of a MDPN and the reward structure. The MDP can be obtained from the RG of the PN model in two steps: (1) build from the RG the RG_{nd} such that given any non deterministic state nd and any probabilistic state pr all maximal non deterministic sub-paths from nd to pr are reduced to a single non deterministic step; (2) build the RG_{MDP} (*i.e.*, a MDP) from the RG_{nd} such that given any non deterministic state nd and any probabilistic state pr, all maximal probabilistic sub-paths from pr to nd are substituted by a single probabilistic step. Finally derive the MDP reword from rs,rt and rg functions.

Let nd be a non deterministic state reached by a probabilistic transition (such states will be the non deterministic states of RG_{nd}). We focus on the subgraph "rooted" in nd and obtained by the maximal non deterministic paths starting from nd. Note that the probabilistic states occurring in this subgraph are terminal states. If there is no finite maximal non deterministic sub-paths starting from nd then no probabilistic phase can follow. So the construction is aborted. Otherwise, given every probabilistic state pr of the subgraph, one wants to obtain the optimal path $\sigma_{nd,pr}$ from nd to pr w.r.t. the reward. Once for every such pr, this path is computed, in RG_{nd} an arc is added from nd to pr labeled by $\sigma_{nd,pr}$. The arcs starting from probabilistic states are unchanged in RG_{nd}.

Thus the building of RG_{nd} depends on whether the optimization problem is a maximization or a minimization of the reward. We only explain the minimization case (the other case is similarly handled). We compute such a sequence using the Bellman and Ford (BF) algorithm for a single-source shortest paths in a weighted digraph where the transition reward is the cost function associated with the arcs. This algorithm is sound due to our (cumulative) definition for rewards of transition sequences. Note that if the BF algorithm finds a negative loop (*i.e.*, where the reward function decreases), the translation is aborted. Indeed the optimal value is then $-\infty$ and there is no optimal sequence: this problem must be solved at the design level.

We now explain how to transform RG_{nd} into the MDP RG_{MDP}. Given a probabilistic state pr and a non deterministic state nd we want to compute the probability to reach nd along probabilistic sub-paths. Furthermore, the sum of these transition probabilities over non deterministic states must be 1. So if in RG_{nd}, there is a terminal strongly connected component composed by only probabilistic states, we abort the construction. The checked condition is necessary and sufficient according to Markov chain theory. Otherwise, we obtain the transition probabilities using two auxiliary matrices. $\mathbf{P}^{(pr,pr)}$, a square matrix indexed by the probabilistic states, denotes the one-step probability transitions between these states and $\mathbf{P}^{(pr,nd)}$, a matrix whose rows are indexed by the probabilistic states and columns are indexed by non deterministic states, denotes the one-step probability transitions from probabilistic states to non deterministic ones. Let us describe how these transition probabilities are obtained. These probabilities are obtained by normalizing the weights of the transitions enabled in pr. Now again, according to Markov chain theory, matrix $\mathbf{P} = (\mathbf{Id} - \mathbf{P}^{(pr,pr)})^{-1} \circ \mathbf{P}^{(pr,nd)}$, where $\mathbf{Id}$ is the identity matrix represents the searched probabilities. A similar transformation is performed in the framework of stochastic Petri nets with immediate transitions (see [12] for the details).

Finally in the MDP, the probability distribution $p(\cdot|nd, \sigma)$ associated with state nd and (complex) action σ, assuming $nd \xrightarrow{\sigma} pr$, is given by the row vector $\mathbf{P}[pr, \cdot]$ and the reward function for every pair of state and action is defined by the following formula: $r(nd, \sigma) = rg(rs(nd), rt(\sigma))$. Since rg is not decreasing w.r.t. its second parameter, the optimal path w.r.t. rt found applying the Bellman and Ford algorithm is also optimal w.r.t. $rg(rs(nd), rt(\cdot))$.

Discussion The MDPN is a high-level formalism for specifying MDPs. However this formalism suffers a drawback: by definition, the components are identified and always distinguished in the state representation, even if they have similar behavior (*i.e.*, even if one component is an exact copy of another component). This can have an impact both at the level of the model description (which could become difficult to read when several components are present), and at the level of the state space size. In the next section, we cope with these problems by introducing a higher-level formalism.

3 Markov Decision Well-Formed Net

3.1 WN Informal Introduction

WNs are an high-level Petri net formalism whose syntax has been the starting point of numerous efficient analysis methods. Below, we describe the main features of WNs. The reader can refer to [4] for a formal definition.

In a WN (and more generally in high-level nets) a color domain is associated with places and transitions. The colors of a place label the tokens contained in this place, whereas the colors of a transition define different ways of firing it. In order to specify these firings, a color function is attached to every arc which, given a color of the transition connected to the arc, determines the number of colored tokens that will be added to or removed from the corresponding place. The initial marking is defined by a multi-set of colored tokens in each place.

A color domain is a Cartesian product of color classes which may be viewed as primitive domains. Classes can have an associated (circular) order expressed by means of a successor function. The Cartesian product defining a color domain is possibly empty (e.g., for a place which contains neutral tokens) and may include repetitions (e.g., a transition which synchronizes two colors inside a class). A class can be divided into static subclasses. The colors of a class have the same nature (processes, resources, etc.), whereas the colors inside a static subclass have the same potential behavior (batch processes, interactive processes, etc.).

A color function is built by standard operations (linear combination, composition, etc.) on basic functions. There are three basic functions: a projection which selects an item of a tuple and is denoted by a typed variable (e.g., p, q); a synchronization/diffusion that is a constant function which returns the multiset composed by all the colors of a class or a subclass and is denoted S_{C_i} ($S_{C_{i,k}}$) where C_i ($C_{i,k}$) is the corresponding (sub)class; and a successor function which applies on an *ordered* class and returns the color following a given color.

Transitions and color functions can be guarded by expressions. An expression is a boolean combination of atomic predicates. An atomic predicate either identifies two variables $[p = q]$ or restricts the domain of a variable to a static subclass.

Examples of arc functions, transition guards, color domains can be seen in the model portions of Fig. 3 and Fig. 4. The details about the WN notation can be found in [4].

The constraints on the syntax of WN allow the automatic exploitation of the behavioral symmetries of the model and the performance of the state-space based analysis on a more compact RG: the symbolic reachability graph (SRG). The SRG construction lies on the *symbolic marking* concept, namely a compact representation for a set of equivalent ordinary markings. A symbolic marking is a symbolic representation, where the actual identity of tokens is forgotten and only their distributions among places are stored. Tokens with the same distribution and belonging to the same static subclass are grouped into a so-called dynamic subclass. Starting from an initial symbolic marking, the SRG can be constructed automatically using a symbolic firing rule [4].

Various behavioral properties may be directly checked on the SRG. Furthermore, this construction leads also to efficient quantitative analysis, e.g. the performance evaluation of Stochastic WNs (SWNs) [4] (a SWN is obtained from a WN by associating an exponentially distributed delay with every transition, which may depend only on the static subclasses to which the firing colors belong).

3.2 Markov Decision Well-Formed Net Definition

A Markov Decision Well-formed Net, like an MDPN, is composed by two distinct parts: the probabilistic one and the non deterministic one, and also in this case the set of transitions in each part is partitioned into $Trun$ and $Tstop$. Each part of a MDWN is a WN model: the two parts share the same set of color classes. A MDWN comprises a special color class, say C_0, representing the system components: its cardinality $|C_0|$ gives the total number of components in the system. This class can be partitioned into several static subclasses $C_0 = (\biguplus_{k=1}^{m} C_{0,k}) \uplus (\biguplus_{k=m+1}^{n_0} C_{0,k})$ such that colors belonging to different static subclasses represent components with different behavior and the first m static subclasses represent the controllable components while the others represent the non-controllable components. Observe that the model is parametric in the number of components of each.

Let us describe the specification of transition triggering by components in an MDWN. First, remember that the firing of a transition t involves the selection a color $c = (c_{i,j})_{i \in 0..n, j \in 1..e_i} \in cd(t) = \bigotimes_{i \in 0..n} C_i^{e_i}$. Thus the subset of components $\{c_{0,j}\}_{j \in 1..e_0}$ defines which components trigger the firing of $t(c)$.

- When the *type* ($synctype(t)$) of t is *Some* then the subset of components that trigger this firing is $Comp(t,c) = \{c_{0,j}\}_{j \in dyn(t)}$, where $dyn(t) \subseteq \{1,\ldots,e_0\}$. Note that when t is a probabilistic transition, this requires that $dyn(t) \neq \emptyset$ whereas when t is a non deterministic one, this requires that $|dyn(t)| \leq 1$ (with the convention that $dyn(t) = \emptyset$ means that t is a decision relative to the system). Furthermore in the latter case, we assume that the guard of t entails that when $dyn(t) = \{c_{0,j}\}$, $c_{0,j} \in \biguplus_{k=1}^{m} C_{0,k}$, *i.e.* $c_{0,j}$ is a controllable component.
- When the *type* of t is *Allbut* then the subset of components that trigger this firing is $Comp(t,c) = \biguplus_{k \in static(t)} C_{0,k} \setminus \{c_{0,j}\}_{j \in dyn(t)}$ where $static(t) \subseteq \{1,\ldots,n_0\}$. Note that this type requires that t is a probabilistic transition. Additional conditions in the following definition ensure that this set of components is not empty.

Definition 5 (Markov Decision Well-formed Net (MDWN)). *A Markov Decision Well-formed is a tuple* $\mathcal{MDWN} = \langle N^{pr}, N^{nd}, synctype, dyn, static\rangle$ *where:*

- N^{pr} *is defined by a WN* $\langle P, T^{pr}, \mathcal{C}, cd^{pr}, I^{pr}, O^{pr}, H^{pr}, \phi^{pr}, prio^{pr}, m_0\rangle$, *a mapping weights for each transition t, from* $cd^pr(t)$ *to* $\mathbb{R}$
- N^{nd} *is defined by a WN* $\langle P, T^{nd}, \mathcal{C}, cd^{nd}, I^{nd}, O^{nd}, H^{nd}, \phi, prio, m_0\rangle$;
- $synctype : T^{pr} \cup T^{nd} \rightarrow \{Some, Allbut\}$ *is a function which associates with every transition a label, s.t.* $\forall t \in T^{nd} \Rightarrow synctype(t) = Some$.
- $dyn(t)$, *where* $t \in T^{pr} \cup T^{nd}$ *and* $cd(t) = \bigotimes_{i \in \{0,\ldots,n\}} C_i^{e_i}$, *is a subset of* $\{1,\ldots,e_0\}$ *(cd is either* cd^{pr} *or* cd^{nd}*);*
- $static(t)$, *defined when* $synctype(t) = Allbut$, *is a subset of* $\{1,\ldots,n_0\}$ *where* n_0 *represents the number of static subclasses in* C_0.

Furthermore, the following constraints must be fulfilled:

- $T^{pr} \cap T^{nd} = \emptyset$;
- $T^{pr} = Trun^{pr} \uplus Tstop^{pr} \wedge T^{nd} = Trun^{nd} \uplus Tstop^{nd}$;
- $\forall t \in T^{pr} \wedge synctype(t) = Some \Rightarrow dyn(t) \neq \emptyset$;
- $\forall t$ *s.t.* $synctype(t) = Allbut$, $\sum_{j \in static(t)} |C_{0,j}| > |dyn(t)|$ *(see discussion above);*
- $\forall t \in T^{nd} \Rightarrow 0 \leq |dyn(t)| \leq 1$; *moreover the transition guard* $\phi(t)$ *should enforce that when* $t(c)$ *is fireable with* $c = (c_{i,k})_{i \in 0..n, k \in 1..e_i} \in cd(t)$ *and* $j \in dyn(t)$ *then* $c_{0,j} \in \biguplus_{k=1}^{m} C_{0,k}$;
- $\forall c_0 \in C_0, \exists t \in Tstop^{pr}, \exists c \in cd(t)$, *s.t.* $\phi(t)(c) \wedge c_0 \in Comp(t,c)$ *and* $\forall c_0 \in \biguplus_{k=1}^{m} C_{0,k}, \exists t \in Tstop^{nd}, \exists c \in cd(t)$, *s.t.* $\phi(t)(c) \wedge c_0 \in Comp(t,c)$. *These conditions can be ensured by appropriate syntactical sufficient conditions.*
- $\forall \{j, j'\} \subseteq dyn(t) \wedge j \neq j', \forall c = (c_{i,k})_{i \in 0..n, k \in 1..e_i} \in cd(t)$ *s.t.* $t(c)$ *is possibly fireable one must have* $c_{0,j} \neq c_{0,j'}$. *This should be enforced by the transition guard.*

Now we introduce the rewards associated to the MDWN. Two types of reward functions are possible: the place reward and the transition reward. Before introducing the place reward we must define the set $\widetilde{C}$.

Definition 6 **($\widetilde{\mathcal{C}}$).** *Let $i \in \{1, \ldots, n\}$, $\widetilde{C}_i$ is the set $\{1, \ldots, n_i\}$ where n_i is the number of static subclasses in C_i. $\widetilde{\mathcal{C}}$ is the set of sets $\{\widetilde{C}_i\}_{i \in I}$ with $I = \{0, \ldots, n\}$.*

We can always map the color class $\mathcal{C}$ on the set $\widetilde{\mathcal{C}}$ such that the definition of the $\widetilde{cd}$ function immediately follows.

Definition 7 **($\widetilde{cd}$).** *The function $\widetilde{cd(p)}$ is defined as follows:*

$$\widetilde{cd} \overset{def}{=} (\widetilde{\bigotimes_{i \in I} C_i^{e_i}}) = \bigotimes_{i \in I} \widetilde{C}_i^{e_i}$$

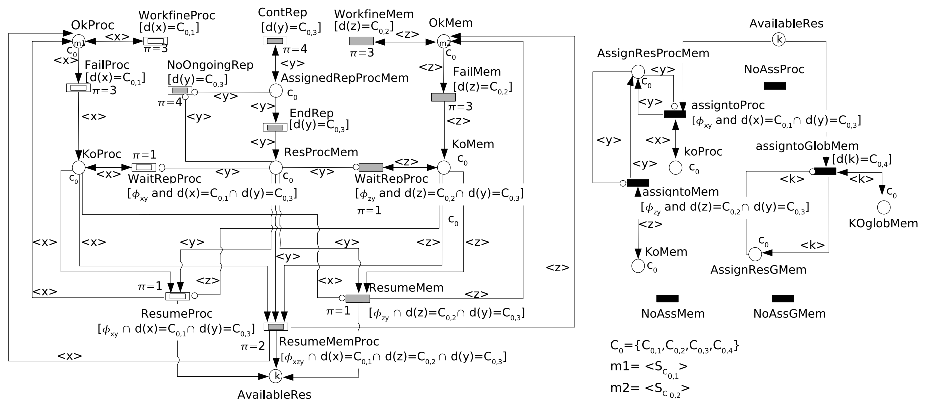
For instance if $C_0 = C_{0,1} \cup C_{0,1} \cup C_{0,3}$ where $C_{0,1} = \{comp_1, comp_2\}, C_{0,2} = \{comp_3\}, C_{0,3} = \{comp_4\}$, hence $\tilde{C}_0 = \{C_{0,1}, C_{0,1}, C_{0,3}\}$, and $p \in P$ with $cd(p) = C_0 \times C_0 \times C_0$ then $\widetilde{cd}(p) = \widetilde{C}_0 \times \widetilde{C}_0 \times \widetilde{C}_0$, $c = \langle comp_1, comp_2, comp_3 \rangle \in cd(p)$ and $\widetilde{c} = \langle 1, 1, 2 \rangle \in \widetilde{cd}(p)$.

It is important to observe that a unique $\widetilde{c}$ corresponds to every c.

Definition 8 **(MDWN reward functions).**

- $rs : \bigotimes_{p \in P} \mathbb{N}^{\widetilde{cd}(p)} \rightarrow \mathbb{R}$ *is a function which returns for every colored marking a reward value.*
- $\forall t \in T^{nd}, rt[t] : cd(t) \rightarrow \mathbb{R}$ *is a vector which associates with every transition a function defining the reward value of its instances; two instances may be assigned a different reward value only if there exists a standard predicate capable to distinguish the two.*
- $rg : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ *is defined as in MDPN.*

An example of MDWN is shown in Fig. 3, the same already used to illustrate MDPNs, but this time color annotations on arcs, transitions and places are relevant. In this model we are assuming that there are several instances of Proc, Mem and ResCtr components (grouped in banks, each with one instance of each component): rather than replicating the same subnet several times, we use colored tokens to represent several instances on the same net structure (there is also another controllable component not shown in the probabilistic subnet, but visible in the decision maker). Class C_0 comprises four static subclasses, one for each component type. The cardinality of the Proc, Mem and ResCtr subclasses corresponds to the number of banks in the system. Arcs in Fig. 3 are annotated with very functions (tuples of projections) and all the variables appearing in the functions in this example are component parameters. The guards on the arcs include a term in the form $d(x) = CompType$ to force parameter x to range within static subclass $CompType$. The additional terms ϕ_{xyz}, ϕ_{xz}, ϕ_{yz} are not detailed here, but are used to associate components in the same bank: in fact the probabilistic part of the model must correctly synchronize components of type Proc, Mem and ResCtr belonging to the same bank (the model represents a situation where only one resource is assigned to each bank at a time, and it can be used to resume all failed components in the bank).



$Trun^{pr} = \{FailProc, FailMem, EndRep\}$ all other transitions belong to $Tstop^{pr}$; all variables are component parameters. All transitions in the decision maker are in $Tstop^{nd}$. Transition priorities are denoted $\pi = prio(t)$ in the figure. $C_{0,1}$, $C_{0,2}$ and $C_{0,3}$ are the Proc, Mem and ResCtr subclasses respectively. Here we have represented the probabilistic transitions with different simbols (double rectangle, gray rectangle and double gray rectangle) depending on the involved components

Fig. 3. MDWN example. On the left: a portion of probabilistic part, on the right: the decision maker.

3.3 MDWN Semantics

In this section we are going to describe how it is possible to obtain from an MDWN model the corresponding MDP model. The two possible methods are shown in the figure of the introduction.

The first method requires the unfolding of the MDWN in order to obtain an equivalent MDPN and to derive from this an MDP, but this is not very efficient in fact it will multiply the number of places, transitions and arcs, moreover if the number of components is high the cost for computing the results will be high. In [2] it is possible to find the details of this method.

Instead the second method derives directly from an MDWN model an MDP. This second method can be decomposed in two steps: the first step defines how to compose the probabilistic part and the decision maker and to derive from such composition a unique WN. The second step consists in generating the (finite) RG of the WN obtained in the first step and then in deriving an MDP from it. In this way there is no need to produce the intermediate MDPN.

Before describing the second method we must explain the use of the places $Stop_l^{pr}$, Run_l^{pr}, $Stop_l^{nd}$, Run_l^{nd}, $Stop_g^{nd}$, Run_g^{nd} and the non deterministic transitions $PrtoNd$ and $NdtoPr$, that are introduced during the composition phase.

The places $Stop^{pr}$, Run^{pr}, $Stop_l^{nd}$, Run_l^{nd}, $Stop_g^{nd}$ and Run_g^{nd} are used in order to regulate the interaction among the components, the global system and the decision maker like the similar places in the semantics for MDPN. The color domain of the places $Stop^{pr}$, Run^{pr}, $Stop_l^{nd}$ is C_0, that is they will contain colored tokens representing the components; while Run_g^{nd}, $Stop_g^{nd}$ are neutral.

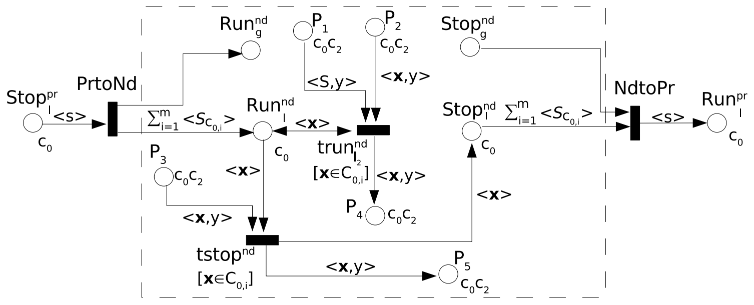


Fig. 4. arcs connecting places $Stop^{pr}$, Run^{nd}_l, Run^{nd}_g, and transition $PrtoNd$ and their functions; arcs connecting places $Stop^{nd}_l$, $Stop^{nd}_g$, Run^{nd}_l and transition $NdtoPr$ and their function; example of connection of the decision maker to places Run^nd and $Stop^nd$: component parameters are highlighted in boldface in the arc functions.

The non deterministic transitions $PrtoNd$ and $NdtoPr$ are used to assure that the decision maker takes a decision for every component in every time unit.

The schema describing how the places $Stop^{pr}$, Run^{pr}, $Stop^{nd}_l$, Run^{nd}_l, $Stop^{nd}_g$ and Run^{nd}_g and the transitions $PrtoNd$ and $NdtoPr$ are connected, is shown in Fig.4. Observe that the basic schema is the same already defined for MDPN but now the arcs are annotated with function $< S >$ meaning that all components must synchronize at that point.

Let us describe how to derive a unique WN composing the probabilistic part with the non deterministic part. Places Run^{pr} and $Stop^{pr}$, introduced above, are connected with its run/stop transitions of N^{pr} in the same way as for MDPNs, similarly places Run^{nd}_l and $Stop^{nd}_l$ Run^{nd}_g and $Stop^{nd}_g$ introduced above are connected to the run/stop transitions of N^{nd} as for MDPNs, but now the arcs must be annotated with the following functions.

- $\forall t \in T^{pr} \cup T^{nd}_l$, if $synctype(t) = Some$ then the function is $\langle \sum_{i \in dyn(t)} x_{0,i} \rangle$, where variable $x_{0,i}$ denotes the i-th component of type C_0 in the color domain of t. This function selects the colors of the component that trigger the transition, thus checking that all of them are still active.
- $\forall t \in Trun^{pr}$, if $synctype(t) = Allbut$ then the function is $\langle \sum_{j \in static(t)} S_{0,j} - \sum_{i \in dyn(t)} x_{0,i} \rangle$ with the same interpretation.

Observe that the arcs connecting transitions T^{nd}_g and places Run^{nd}_g, $Stop^{nd}_g$ are not annotated with any function because these places have *neutral* color (i.e. they contain plain black tokens) since they are related to the decision w.r.t. the whole system.

Once the composed WN is built, its RG can be constructed and transformed into a MDP following the same two steps already explained for MDPN. Here, since we start from a high-level net, the resulting reachability graph may be huge. So the following subsection describe how the properties of WN can be extended to MDWN so that a smaller MDP can be directly derived from the Symbolic Reachability Graph (SRG) of the corresponding WN.

3.4 Theoretical Results on Symmetry Exploitation

In this section, we informally describe how we exploit the symbolic reachability graph in order to obtain a reduced MDP on which the solution to the original problem can be computed (see [2] for a complete theoretical development).

First, let us pick a symbolic reachable marking which only enables non deterministic transitions and an ordinary marking belonging to this symbolic marking. Now let us pick two ordinary firings from this marking corresponding to the same symbolic firing. Suppose that, at some instant of an execution, a strategy selects one of these firings. Then, after selecting the other firing, one mimics the original strategy by applying one of the permutations which lead from the former firing to the latter one to any subsequent (probabilistic or non deterministic) firing and let invariant the ordinary marking. Due to our assumptions about the rewards, the two executions yield the same (total or average) reward. It means that the choice of the second firing is at least as good as the selection of the first firing. Since the argument is symmetric, one concludes that the selection of any non deterministic firing inside a symbolic arc is irrelevant.

Then the reduced MDP obtained from the SRG by considering that a symbolic firing of a non deterministic transition corresponds to a *single* decision and that the weight of probabilistic symbolic firing is the weight of any ordinary firing inside it (any choice leads to the same weight due to our assumptions) multiplied by the number of such firings provides an MDP *equivalent* to the original one w.r.t. the considered optimization problem. Indeed the rewards do not depend on the choice of an ordinary marking inside a symbolic marking and the choice of an ordinary firing inside a symbolic firing. We will call SRG_{nd} the SRG where all the transition instances passing only through non deterministic states are reduced to one non deterministic step and SRG_{MDP} the SRG_{nd} where all probabilistic paths are substituted by single probabilistic arcs.

4 Experiments Discussion

In this section we will present an example modeling a multiprocessor system where each processor has a local memory, but with also a global shared memory that can be used by any processor when its local memory fails. Each processor, local memory and global shared memory can fail independently; however we consider recoverable failures, that can be solved by restarting/reconfiguring the failed component. The system includes an automatic failure detection system that is able to detect and perform a reconfiguration of the failed component (e.g. by resetting it). The failure detection and recovery system can handle a limited number k of failures in parallel.

Notice that if a local memory M_i and the global shared memory Mg are both failed at the same time, the processor P_i cannot perform any useful work,

Table 1. Results for the example modeling a multiprocessor system. The RG for Proc=4 and Mem=4 is not computed because it requires a lot of time; its size is computed indirectly by the SRG.

	Proc=2,Mem=2,Res=2			Proc=3,Mem=3,Res=2			Proc=4,Mem=4,Res=2		
	Prob.	Non det.	Time	Prob.	Non det.	Time	Prob.	Non det.	Time
RG	19057	21031	13s	755506	863886	1363s	26845912	31895848	>13h
RG_{nd}	19057	441	9s	755506	4078	2833s			
RG_{MDP}	0	441	2s	0	4078	250s			
SRG	9651	10665	9s	132349	150779	284s	1256220	1478606	5032s
SRG_{nd}	9651	219	3s	132349	831	222s	1256220	2368	12795s
SRG_{MDP}	0	219	1s	0	831	28s	0	2360	518s
RG **prio**	19057	5235	9s	755506	103172	983s	26845912	1863024	>13h
RG_{nd} **prio**	19057	411	4s	755506	4078	1830s			
RG_{MDP} **prio**	0	411	2s	0	4078	246s			
SRG **prio**	9651	2697	6s	132349	18904	187s	1256220	96044	3270s
SRG_{nd} **prio**	9651	219	2s	132349	831	75s	1256220	2368	1560s
SRG_{MDP} **prio**	0	219	1s	0	831	26s	0	2360	234s

even if it is not failed and that if the processor P_i and its local memory M_i are simultaneously failed, they are reset together (this is considered as a single reset operation). The components in this system are: n processors, n local memories and one global shared memory. A portion of MDWN representation of this system is depicted in Fig. 3

The decision maker corresponds to the automatic failure detection and recovery system. Several different recovery strategies can be conceived, and we are interested in evaluating the most promising ones with respect to some metrics.

An MDPN (or MDWN) model of this system is composed of a submodel representing all the components of the system (which in turn can be seen as a combination of several submodels of the single components), and a submodel representing the failure detection and recovery system, which in this context corresponds to the decision maker.

The decision maker model may represent any possible recovery strategy, in this case it should be modeled in such a way that any association of up to k recovery resources to any subset of failed components at a given time can be realized by the model.The system must pay a penalty depending of the number of running processors when the number of running processors is less than a given threshold and a repair cost for every recovery. More details about this example are shown in [2]. The table 1 shows the number of states and the computation time respectively of the $RG, RG_{nd}, RG_{MDP}, SRG, SRG_{nd}$ and SRG_{MDP} for different numbers of processors and memories performed with an AMD Athlon 64 2.4Ghz of 4Gb memory capacity. In particular the first, the second and the third line report the number of states and computation time of the RG, the RG_{nd} and the RG_{mdp}, while the following three lines show the number of states and the computation time obtained using the SRG technique. It is easy to observe how the SRG technique wins in terms of memory and time gain with respect to the RG technique.

A further reduction of the number of states for this model can be achieved associating different priorities to the system transitions such that the interleavings between the non deterministic/probabilistic actions are reduced. For instance the last six lines in table 1 show the reduction in terms of non deterministic states and time computation obtained imposing an order on the decision maker choices. (First the decision maker must take all the decisions for the processors then for the memories and in the end for the global memory).

It is not always possible to use this trick since the actions must be independent; the priorities in practice must not reduce the set of possible strategies. Our tool solves the MDPs using the library *graphMDP* developed at the Ecole Nationale Suprieure de l'Aronautique et de l'Espace Toulouse. The optimal strategy is expressed as a set of optimal actions, such that for every system state an optimal action is given.

For example if we consider a model with two processors, two memories and two recovery resources, and with reasonable fault probability, and repair and penalty costs then we observe that if a fault happens and there is a free recovery resource then the recovery of this fault starts immediately and the global memory recovery is preferred with respect the processor recovery and the memory recovery.This is not always true, e.g. if global memory recovery cost is more than four times of the memory repair cost.

After having obtained the optimal strategy we would like to synthesize a new model without non determinism implementing it (this could be achieved by substituting the decision maker part with a new probabilistic part implementing the decisions of the optimal strategy): classical Markov chain analysis techniques could be applied to this model, moreover the new net would constitute a higher level (hopefully easier to interpret) description for the optimal strategy. Unfortunately this is not always easy (especially when the number of states is large), but this is an interesting direction of future research.

5 Related Work

In this section we are going to compare our formalism with two other high level formalisms for MDP: the PRISM language and the Stochastic Transition System (STS) proposed in [8].

The PRISM language [11] is a state-based language based on the Reactive Modules formalism of Alur and Henzinger [1]. A system is modeled by PRISM language as composition of modules(components) which can interact with each other. Every model contains a number of local variables used to define it state in every time unit, and the local state of all modules determines the global state. The behavior of each module is described by a set of commands; such that a command is composed by a guard and a transition. The guard is a predicate over all the (local/nonlocal) variables while a transition describes how the local variable will be update if the its guard is true.

The composition of the modules is defined by a process-algebraic expression: parallel composition of modules, action hiding and action renaming.

Comparing the MDPN formalism with the PRISM language we can observe that they have the same expressive power: we can define local or global non-deterministic actions and the reward function on the states and/or on the actions in both formalisms; such that it is possible to translate MDPN model directly in a PRISM model. The main difference is that by using the MDPN formalism one can define complex probabilistic behaviors and complex non-deterministic actions as a composition of simpler behaviors or actions.

If we compare the PRISM language with the MDWN then we can see that the MDWN has two other advantages: a parametric description of the model and an efficient analysis technique making it possible to automatically take advantage of intrinsic symmetries of the system. In fact the PRISM language has a limited possibility for parametrization. In order to cope with this problem in [9] it was presented a syntactic pre-processor called eXtended Reactive Modules (XRM) which can generate RM models giving to the users the possibility of describing the system using for instance: *for* loops, *if* statements.

Instead several techniques proposed in order to reduce the states explosion problem in PRISM i.e. in [10] were based on the minimization of the RG with respect to bisimulation; but this requires the building of all the state space and then to reduce it; hence our method gives the possibility of managing models with a bigger number of states. It generates directly the Lumped MDP without building all the state space.

A direct comparison between our formalisms and the STS is not possible, because the STSs are an high level formalism for modeling the continuous time MDPs. It extends the Generalized Stochastic Petri Net by introducing transitions with an unspecified delay distributions and by the introducing the possibility of non-deterministic choice among enabled immediate transitions. In every way we can observe that the STS has the same problems of GSPN formalism; that make its utilization less advantageous with respect to the WN. It is also important to observe that there are no tools supporting this formalism.

6 Conclusion

We have introduced MDPNs, based on Petri nets, and MDWNs, based on Well-formed nets, in order to model and analyze distributed systems with probabilistic and non deterministic features. From a modeling point of view, these models support a macroscopic point of view of alternation between the non probabilistic behavior and the non deterministic one of the system and a syntactical way to define the switch between the two behaviors. Furthermore, MDWNs enable the modeler to specify in a concise way similar components. From an analysis point of view, we have adapted the technique of the symbolic reachability graph producing a reduced Markov decision process w.r.t. the original one, on which the analysis may be performed. Our methods are already implemented and integrated in the GreatSPN tool and we have described some experimental results.

References

1. Alur, R., Henzinger, T.: Reactive modules. Formal Methods in System Design 15(1), 7–48 (1999)
2. Beccuti, M., Franceschinis, G., Haddad, S.: Markov Decision Petri Net and Markov Decision Well-formed Net formalisms. Technical Report TR-INF-2007-02-01, Dipartimento di Informatica, Università del Piemonte Orientale (2007) http://www.di.unipmn.it/Tecnical-R
3. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
4. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic well-formed coloured nets for symmetric modelling applications. IEEE Transactions on Computers 42(11), 1343–1360 (November 1993)
5. 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)
6. Chiola, G., Franceschinis, G., Gaeta, R., Ribaudo, M.: GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic petri nets. Performance Evaluation, special issue on Performance Modeling Tools 24(1-2), 47–68 (November 1995)
7. de Alfaro, L.: Temporal logics for the specification of performance and reliability. In: Reischuk, R., Morvan, M. (eds.) STACS 97. LNCS, vol. 1200, pp. 165–176. Springer, Heidelberg (1997)
8. de Alfaro, L.: Stochastic transition systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 423–438. Springer, Heidelberg (1998)
9. Demaille, K., Peyronnet, S., Sigoure, B.: Modeling of sensor networks using XRM. In: 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Paphos, Cyprus (2006)
10. Garavel, H., Hermanns, H.: On combining functional verification and performance evaluation using CADP. In: FME 2002. LNCS, vol. 2391, pp. 10–429. Springer, Heidelberg (2000)
11. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
12. Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. Wiley Series in Parallel Computing. John Wiley and Sons, New york (1995) http://www.di.unito.it/~greatspn
13. Puterman, M.L.: Markov Decision Processes. In: Discrete Stochastic Dynamic Programming, Wiley, Chichester (2005)

Comparison of the Expressiveness of Arc, Place and Transition Time Petri Nets

M. Boyer[1] and O.H. Roux[2]

[1] IRIT Toulouse France
IRIT/ENSEEIHT, 2 rue Camichel, BP 7122, 31071 Toulouse Cedex 7
marc.boyer@enseeiht.fr
[2] IRCCyN, Nantes, France
1 rue de la Noë 44321 Nantes Cedex 3 France
olivier-h.roux@irccyn.ec-nantes.fr

Abstract. In this paper, we consider bounded Time Petri Nets where time intervals (strict and large) are associated with places (*P-TPN*), arcs (*A-TPN*) or transitions (*T-TPN*). We give the formal strong and weak semantics of these models in terms of Timed Transition Systems. We compare the expressiveness of the six models w.r.t. (weak) timed bisimilarity (behavioral semantics). The main results of the paper are : (i) with strong semantics, *A-TPN* is strictly more expressive than *P-TPN* and *T-TPN* ; (ii) with strong semantics *P-TPN* and *T-TPN* are incomparable ; (iii) *T-TPN* with strong semantics and *T-TPN* with weak semantics are incomparable. Moreover, we give a classification by a set of 9 relations explained in Fig. 14 (p. 80).

1 Introduction

The two main extensions of Petri Nets with time are Time Petri Nets (TPNs) [18] and Timed Petri Nets [20]. For TPNs a transition can fire within a time interval whereas for Timed Petri Nets it has a duration and fires as soon as possible or with respect to a scheduling policy, depending on the authors. Among Timed Petri Nets, time can be considered relative to places (P-Timed Petri Nets), arcs (A-Timed Petri Nets) or transitions (T-Timed Petri Nets) [21,19]. The same classes are defined for TPNs i.e. *T-TPN* [18,5], *A-TPN* [14,1,13] and *P-TPN* [16,17]. It is known that P-Timed Petri Nets and T-Timed Petri Nets are expressively equivalent [21,19] and these two classes of Timed Petri Nets are included in the two corresponding classes *T-TPN* and *P-TPN* [19]

Depending on the authors, two semantics are considered for $\{T,A,P\}$-*TPN*: a weak one, where no transition is never forced to be fired, and a strong one, where each transition must be fired when the upper bound of its time condition is reached. Moreover there are a single-server and several multi-server semantics [8,4]. The number of *clocks* to be considered is finite with single-server semantics (one clock per transition, one per place or one per arc) whereas it is not with multi-server semantics.

A-TPN have mainly been studied with weak (lazy) multi-server semantics [14,1,13] : this means that the number of clocks is not finite but the firing of transitions may be delayed, even if this implies that some transitions are disabled because their input tokens become too old. The reachability problem is undecidable for this class of *A-TPN* but thanks to this weak semantics, it enjoys *monotonic* properties and falls into a class of models for which coverability and boundedness problems are decidable.

Conversely *T-TPN* [18,5] and *P-TPN* [16,17] have been studied with strong single-server semantics. They do not have monotonic features of weak semantics although the number of *clocks* is finite. The marking reachability problem is known undecidable [15] but marking coverability, k-boundedness, state reachability and liveness are decidable for bounded *T-TPN* and *P-TPN* with strong semantics.

Related work : Expressiveness of models extended with time.

Time Petri Nets versus Timed Automata. Some works compare the expressiveness of Time Petri Nets and Timed Automata. In [22], the author exposes mutual isomorphic translations between 1-safe Time-Arc Petri nets (*A-TPN*) and networks of Timed Automata.

In [3,10] it was proved that bounded *T-TPN* with strong semantics form a strict subclass of the class of timed automata wrt timed bisimilarity. Authors give in [10] a characterisation of the subclass of timed automata which admit a weakly timed bisimilar *T-TPN*. Moreover it was proved in [3] that bounded *T-TPN* and timed automata are equally expressive wrt timed language acceptance.

Arc, Place and Transition Time Petri Nets. The comparison of the expressiveness between *A-TPN*, *P-TPN* and *T-TPN* models with strong and weak semantics wrt timed language acceptance and timed bisimulation have been very little studied[1].

In [12] authors compared these models w.r.t. language acceptance. With strong semantics, they established $P\text{-}TPN \subseteq_{\mathcal{L}} T\text{-}TPN \subseteq_{\mathcal{L}} A\text{-}TPN$[2] and with weak semantics the result is $P\text{-}TPN =_{\mathcal{L}} T\text{-}TPN =_{\mathcal{L}} A\text{-}TPN$.

In [6] authors study only the strong semantics and obtain the following results: $T\text{-}TPN \subset_{\mathcal{L}} A\text{-}TPN$ and $P\text{-}TPN \not\subseteq_{\mathcal{L}} T\text{-}TPN$.

These results of [12] and [6] are inconsistent.

Concerning bisimulation, in [6] (with strong semantics) we have $T\text{-}TPN \subset_{\approx} A\text{-}TPN$, $P\text{-}TPN \subseteq_{\approx} A\text{-}TPN$ and $P\text{-}TPN \not\subseteq_{\approx} T\text{-}TPN$. But the counter-example given in this paper to show $P\text{-}TPN \not\subseteq_{\approx} T\text{-}TPN$ uses the fact that the *T-TPN* 'à la Merlin' cannot model strict timed constraint[3]. This counter example fails if we extend these models to strict constraints.

In [17] *P-TPN* and *T-TPN* are declared incomparable but no proof is given.

Much problems remain open concerning the relations between these models.

[1] Moreover, all studies consider only closed interval constraints, and from results in [6], offering strict constraints makes a difference on expressiveness.

[2] we note $\sim_{\mathcal{L}}$ and $\sim_{\approx}$ with $\sim \in \{\subset, \subseteq, =\}$ respectively for the expressiveness relation w.r.t. timed language acceptance and timed bisimilarity.

[3] The intervals are of the form $[a, b]$ and they can not handle a behavior like "if $x < 1$".

Our Contribution. In this paper, we consider bounded Arc, Place and Transition Time Petri Nets with strict and large timed constraints and with single-server semantics. We give the formal strong and weak semantics of these models in terms of Timed Transition Systems. We compare each model with the two others in the weak and the strong semantics, and also the relations between the weak on strong semantics for each model (see Fig. 14, p. 80). The comparison criterion is the weak timed bisimulation. In this set of 9 relations, 7 are completely covered, and for the 2 others, half of the relation is still an open problem.

The paper is organised as follow: Section 2 gives some "framework" definitions. Section 3 presents the three timed Petri nets models, with strong and weak semantics. Section 4 is the core of our contribution: it lists all the new results we propose. Section 5 concludes.

By lack of space, the details of some proofs and the relations with conflicting results are not presented here, but can be found in [9].

2 Framework Definition

We denote A^X the set of mappings from X to A. If X is finite and $|X| = n$, an element of A^X is also a vector in A^n. The usual operators $+, -, <$ and $=$ are used on vectors of A^n with $A = \mathbb{N}, \mathbb{Q}, \mathbb{R}$ and are the point-wise extensions of their counterparts in A. For a *valuation* $\nu \in A^X, d \in A$, $\nu + d$ denotes the vector $(\nu + d)(x) = \nu(x) + d$. The set of boolean is denoted by $\mathbb{B}$. The set of non negative intervals in $\mathbb{Q}$ is denoted by $\mathcal{I}(\mathbb{Q}_{\geq 0})$. An element of $\mathcal{I}(\mathbb{Q}_{\geq 0})$ is a constraint φ of the form $\alpha \prec_1 x \prec_2 \beta$ with $\alpha \in \mathbb{Q}_{\geq 0}$, $\beta \in \mathbb{Q}_{\geq 0} \cup \{\infty\}$ and $\prec_1, \prec_2 \in \{<, \leq\}$, such that $I = [\![\varphi]\!]$. We let $I^{\downarrow} = [\![0 \leq x \prec_2 \beta]\!]$ be the *downward closure* of I and $I^{\uparrow} = [\![\alpha \prec_1 x]\!]$ be the *upward closure* of I.

Let Σ be a fixed finite alphabet s.t. $\varepsilon \notin \Sigma$ and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, with ε the neutral element of sequence ($\forall a \in \Sigma_\varepsilon : \varepsilon a = a\varepsilon = a$).

Definition 1 (Timed Transition Systems). *A timed transition system (TTS) over the set of actions Σ is a tuple $S = (Q, Q_0, \Sigma, \longrightarrow)$ where Q is a set of states, $Q_0 \subseteq Q$ is the set of initial states, Σ is a finite set of actions disjoint from $\mathbb{R}_{\geq 0}$, $\longrightarrow \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$ is a set of edges. If $(q, e, q') \in \longrightarrow$, we also write $q \xrightarrow{e} q'$. Moreover, it should verify some time-related conditions: time determinism (td), time-additivity (ta), nul delay (nd) and time continuity (tc).*

$$td \equiv s \xrightarrow{d} s' \wedge s \xrightarrow{d} s'' \Rightarrow s' = s'' \qquad ta \equiv s \xrightarrow{d} s' \wedge s' \xrightarrow{d'} s'' \Rightarrow s \xrightarrow{d+d'} s''$$

$$nd \equiv \forall s : s \xrightarrow{0} s \qquad tc \equiv s \xrightarrow{d} s' \Rightarrow \forall d' \leq d, \exists s_{d'}, s \xrightarrow{d'} s_{d'}$$

In the case of $q \xrightarrow{d} q'$ with $d \in \mathbb{R}_{\geq 0}$, d denotes a delay and not an absolute time. A *run* ρ of length $n \geq 0$ is a finite ($n < \omega$) or infinite ($n = \omega$) sequence of alternating time and discrete transitions of the form

$$\rho = q_0 \xrightarrow{d_0} q'_0 \xrightarrow{a_0} q_1 \xrightarrow{d_1} q'_1 \xrightarrow{a_1} \cdots q_n \xrightarrow{d_n} q'_n \cdots$$

A trace of ρ is the timed word $w = (a_0, d_0)(a_1, d_1) \cdots (a_n, d_n) \cdots$ that consists of the sequence of letters of Σ.

We write $Untimed(\rho) = Untimed(w) = a_0 a_1 \cdots a_n \cdots$ for the untimed part of w, and $Duration(\rho) = Duration(w) = \sum d_k$ for the duration of the timed word w and then of the run ρ.

Definition 2 (Strong Timed Bisimilarity). *Let $S_1 = (Q_1, Q_0^1, \Sigma, \longrightarrow_1)$ and $S_2 = (Q_2, Q_0^2, \Sigma, \longrightarrow_2)$ be two TTS*[4] *and $\approx_{\mathcal{S}}$ be a binary relation over $Q_1 \times Q_2$. We write $s \approx_{\mathcal{S}} s'$ for $(s, s') \in \approx_{\mathcal{S}}$. $\approx_{\mathcal{S}}$ is a* timed bisimulation relation *between S_1 and S_2 if:*

- *$s_1 \approx_{\mathcal{S}} s_2$, for all $(s_1, s_2) \in Q_0^1 \times Q_0^2$;*
- *if $s_1 \xrightarrow{t}_1 s_1'$ with $t \in \mathbb{R}_{\geq 0}$ and $s_1 \approx_{\mathcal{S}} s_2$ then $s_2 \xrightarrow{t}_2 s_2'$ for some s_2', and $s_1' \approx_{\mathcal{S}} s_2'$; conversely if $s_2 \xrightarrow{t}_2 s_2'$ and $s_1 \approx_{\mathcal{S}} s_2$ then $s_1 \xrightarrow{t}_1 s_1'$ for some s_1' and $s_1' \approx_{\mathcal{S}} s_2'$;*
- *if $s_1 \xrightarrow{a}_1 s_1'$ with $a \in \Sigma$ and $s_1 \approx_{\mathcal{S}} s_2$ then $s_2 \xrightarrow{a}_2 s_2'$ and $s_1' \approx_{\mathcal{S}} s_2'$; conversely if $s_2 \xrightarrow{a}_2 s_2'$ and $s_1 \approx_{\mathcal{S}} s_2$ then $s_1 \xrightarrow{a}_1 s_1'$ and $s_1' \approx_{\mathcal{S}} s_2'$.*

Two TTS S_1 and S_2 are timed bisimilar *if there exists a timed bisimulation relation between S_1 and S_2. We write $S_1 \approx_{\mathcal{S}} S_2$ in this case.*

Let $S = (Q, Q_0, \Sigma_\varepsilon, \longrightarrow)$ be a TTS. We define the ε-abstract TTS $S^\varepsilon = (Q, Q_0^\varepsilon, \Sigma, \longrightarrow_\varepsilon)$ (with no ε-transitions) by:

- $q \xrightarrow{d}_\varepsilon q'$ with $d \in \mathbb{R}_{\geq 0}$ iff there is a run $\rho = q \xrightarrow{*} q'$ with $Untimed(\rho) = \varepsilon$ and $Duration(\rho) = d$,
- $q \xrightarrow{a}_\varepsilon q'$ with $a \in \Sigma$ iff there is a run $\rho = q \xrightarrow{*} q'$ with $Untimed(\rho) = a$ and $Duration(\rho) = 0$,
- $Q_0^\varepsilon = \{q \mid \exists q' \in Q_0 \mid \ q' \xrightarrow{*} q$ *and* $Duration(\rho) = 0 \wedge Untimed(\rho) = \varepsilon\}$.

Definition 3 (Weak Timed Bisimilarity). *Let $S_1 = (Q_1, Q_0^1, \Sigma_\varepsilon, \longrightarrow_1)$ and $S_2 = (Q_2, Q_0^2, \Sigma_\varepsilon, \longrightarrow_2)$ be two TTS and $\approx_{\mathcal{W}}$ be a binary relation over $Q_1 \times Q_2$. $\approx_{\mathcal{W}}$ is a* weak (timed) bisimulation relation *between S_1 and S_2 if it is a strong timed bisimulation relation between S_1^ε and S_2^ε.*

Note that if $S_1 \approx_{\mathcal{S}} S_2$ then $S_1 \approx_{\mathcal{W}} S_2$ and if $S_1 \approx_{\mathcal{W}} S_2$ then S_1 and S_2 have the same timed language.

In this paper, we consider weak timed bisimilarity and we note $\approx$ for $\approx_{\mathcal{W}}$.

Definition 4 (Expressiveness w.r.t. (Weak) Timed Bisimilarity). *The class $\mathcal{C}$ is* more expressive *than $\mathcal{C}'$ w.r.t. timed bisimilarity if for all $B' \in \mathcal{C}'$ there is a $B \in \mathcal{C}$ s.t. $B \approx B'$. We write $\mathcal{C}' \subseteq_\approx \mathcal{C}$ in this case. If moreover there is a $B \in \mathcal{C}$ s.t. there is no $B' \in \mathcal{C}'$ with $B \approx B'$, then $\mathcal{C}' \subset_\approx \mathcal{C}$. If both $\mathcal{C}' \subset_\approx \mathcal{C}$ and $\mathcal{C} \subset_\approx \mathcal{C}'$ then $\mathcal{C}$ and $\mathcal{C}'$ are equally expressive w.r.t. timed bisimilarity, and we write $\mathcal{C} =_\approx \mathcal{C}'$.*

[4] Note that they contain no ε-transitions.

3 $\{T,A,P\}$-*TPN*: Definitions and Semantics

The classical definition of TPN is based on a single server semantics (see [8,4] for other semantics). With this semantics, bounded-TPN and safe-TPN (ie one-bounded) are equally expressive wrt timed-bisimilarity and then wrt timed language acceptance. We give a proof of this result for $\{T,A,P\}$-*TPN* in [9]. Thus, in the sequel, we will consider safe TPN. We now give definitions and semantics of safe $\{T,A,P\}$-*TPN*.

3.1 Common Definitions

We assume the reader is aware of Petri net theory, and only recall a few definitions.

Definition 5 (Petri Net). *A* Petri Net $\mathcal{N}$ *is a tuple* $(P, T, {}^\bullet(.), (.)^\bullet, M_0, \Lambda)$ *where:* $P = \{p_1, p_2, \cdots, p_m\}$ *is a finite set of* places *and* $T = \{t_1, t_2, \cdots, t_n\}$ *is a finite set of* transitions*;* ${}^\bullet(.) \in (\{0,1\}^P)^T$ *is the* backward *incidence mapping;* $(.)^\bullet \in (\{0,1\}^P)^T$ *is the* forward *incidence mapping;* $M_0 \in \{0,1\}^P$ *is the* initial *marking,* $\Lambda : T \to \Sigma \cup \{\varepsilon\}$ *is the labeling function.*

Notations for all Petri nets
We use the following common shorthands: $p \in M \overset{\text{def}}{=} M(p) \geq 1$, $M \geq {}^\bullet t \overset{\text{def}}{=} \forall p : M(p) \geq {}^\bullet(t,p)$, ${}^\bullet t \overset{\text{def}}{=} \{p \,|\, {}^\bullet(t,p) \geq 1\}$, $t^\bullet \overset{\text{def}}{=} \{p \,|\, (t,p)^\bullet \geq 1\}$, ${}^\bullet p \overset{\text{def}}{=} \{t \,|\, (t,p)^\bullet \geq 1\}$, $p^\bullet \overset{\text{def}}{=} \{t \,|\, {}^\bullet(t,p) \geq 1\}$.

A *marking* M is an element $M \in \{0,1\}^P$. $M(p)$ is the number of tokens in place p. A transition t is said to be *enabled* by marking M iff $M \geq {}^\bullet t$, denoted $t \in$ *enabled*(M). The firing of t leads to a marking $M' = M - {}^\bullet t + t^\bullet$, denoted by $M \xrightarrow{t} M'$.

Often, the alphabet is the set of transitions and the labeling function the identity ($\Sigma = T, \Lambda(t) = t$). In these cases, the label of the transition will not be put in figures.

Notations for all timed Petri nets
In timed extensions of Petri nets, a transition can be fired only if the enabling condition *and* some time related condition are satisfied. In the following, the expressions *enabled* and *enabling* refer only to the marking condition, and *firable* is the conjunction of enabling and the model-specific timed condition.

Then, $t \in$ *firable*(S) denotes that t is firable in timed state S, and $t \in$ *enabled*(M) that t is enabled by marking M.

Weak vs. strong semantics
The basic strong semantics paradigm is expressed in different ways depending on the authors: one expression could be "time elapsing can not disable the firable property of a transition", or "whenever the upper bound of a firing interval is reached, the transition must be fired". Depending on the models and the authors,

this principle is described by different equations. In this paper, the one we are going to use is: a delay d is admissible from state S ([5]) iff

$$t \notin \mathit{firable}(S+d) \Rightarrow \forall d' \in [0,d] : t \notin \mathit{firable}(S+d') \tag{1}$$

which means that from S, if a transition is not firable after a delay d, it never was between S and $S+d$, which is equivalent to say that, if a transition is enabled now or in the future (without discrete transition firing), it remains firable with time elapsing.

3.2 Transition Time Petri Nets (*T-TPN*)

The model. Time Petri Nets were introduced in [18] and extend Petri Nets with timing constraints on the firings of transitions.

Definition 6 (Transition Time Petri Net). *A* Time Petri Net $\mathcal{N}$ *is a tuple* $(P, T, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$ *where:* $(P, T, {}^\bullet(.), (.)^\bullet, M_0, \Lambda)$ *is a Petri net and* $I : T \to \mathcal{I}(\mathbb{Q}_{\geq 0})$ *associates with each transition a* firing interval.

Semantics of Transition Time Petri Nets

The state of *T-TPN* is a pair (M, ν), where M is a marking and $\nu \in \mathbb{R}^T_{\geq 0}$ is a *valuation* such that each value $\nu(t_i)$ is the elapsed time since the last time transition t_i was enabled. $\mathbf{0}$ is the initial valuation with $\forall i \in [1..n], \mathbf{0}(t_i) = 0$.

For Transition Time Petri Net, notations *enabled* and *firable* are defined as follows :

$$t \in \mathit{enabled}(M) \text{ iff } M \geq {}^\bullet t \qquad t \in \mathit{firable}(M,\nu) \text{ iff } \begin{cases} t \in \mathit{enabled}(M) \\ \nu(t) \in I(t) \end{cases}$$

The *newly enabled* function $\uparrow\!\mathit{enabled}(t_k, M, t_i) \in \mathbb{B}$ is true if t_k is enabled by the firing of transition t_i from marking M, and false otherwise. This definition of enabledness is based on [5,2] which is the most common one. In this framework, a transition t_k is *newly enabled* after firing t_i from marking M if "it is not enabled by $M - {}^\bullet t_i$ and is enabled by $M' = M - {}^\bullet t_i + t_i^\bullet$" [5].

Formally this gives:

$$\uparrow\!\mathit{enabled}(t_k, M, t_i) = \left(M - {}^\bullet t_i + t_i^\bullet \geq {}^\bullet t_k\right) \wedge \left((M - {}^\bullet t_i < {}^\bullet t_k) \vee (t_k = t_i)\right) \tag{2}$$

Definition 7 (Strong Semantics of *T-TPN*). *The semantics of a T-TPN* $\mathcal{N}$ *is a timed transition system* $S_\mathcal{N} = (Q, q_0, \to)$ *where:* $Q = \{0,1\}^P \times (\mathbb{R}_{\geq 0})^n$, $q_0 = (M_0, \mathbf{0})$, $\longrightarrow \in Q \times (\Sigma_\varepsilon \cup \mathbb{R}_{\geq 0}) \times Q$ *consists of the discrete and continuous transition relations:*

– *the discrete transition relation is defined* $\forall t \in T$*:*

$$(M,\nu) \xrightarrow{\Lambda(t)} (M',\nu') \text{ iff } \begin{cases} t \in \mathit{firable}(M,\nu) \\ M' = M - {}^\bullet t + t^\bullet \\ \forall t' \in T : \nu'(t') = \begin{cases} 0 & \text{if } \uparrow\!\mathit{enabled}(t', M, t), \\ \nu(t') & \text{otherwise.} \end{cases} \end{cases}$$

[5] The encoding of the state depends on the model.

– *the continuous transition relation is defined* $\forall d \in \mathbb{R}_{\geq 0}$*:*

$$(M,\nu) \xrightarrow{d} (M,\nu') \text{ iff } \begin{cases} \nu' = \nu + d \\ \forall t \in T : t \notin \mathit{firable}(M, v+d) \Rightarrow \\ \qquad\qquad (\forall d' \in [0,d] : t \notin \mathit{firable}(M, v+d')) \end{cases} \tag{3}$$

Definition 8 (Weak Semantics of *T-TPN*). *For safe T-TPN, the only difference of the weak semantics is on the continuous transition relation defined* $\forall d \in \mathbb{R}_{\geq 0}$*:*

$$(M,\nu) \xrightarrow{d} (M,\nu') \text{ iff } \nu' = \nu + d$$

Some examples illustrating the synchronization rule and the difference between weak and strong semantics can be found in [9].

3.3 Place Time Petri Nets (*P-TPN*)

The model. Place Time Petri Nets were introduced in [16], adding interval on places and considering a strong semantics.

Putting interval on places implies that clocks are handled by tokens: a token can be use to fire a transition iff its age in the place is in the interval of the place. A particularity of this model is the notion of *dead token*: a token whose age is greater than the upper bound of its place can never leave this place: it is a *dead token*.

Let *dead* be a mapping in $\{0,1\}^P$. $\mathit{dead}(p)$ is the number of dead tokens in place p ($\forall p \in P : \mathit{dead}(p) \leq M(p)$). We use the following shorthands : $M \backslash \mathit{dead}$ for $M - \mathit{dead}$ and thus $p \in M\backslash \mathit{dead}$ for $M(p) - \mathit{dead}(p) \geq 1$.

Definition 9 (Place Time Petri Net). *A* Place Time Petri Net $\mathcal{N}$ *is a tuple* $(P, T, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$ *where:* $(P, T, {}^\bullet(.), (.)^\bullet, M_0, \Lambda)$ *is a Petri net and* $I : P \to \mathcal{I}(\mathbb{Q}_{\geq 0})$ *associates with each place a* residence time *interval.*

Semantics of Place Time Petri Nets

The state of *P-TPN* is a tuple (M, dead, ν) where M is a marking, *dead* is the dead token mapping and $\nu \in \mathbb{R}_{\geq 0}^M$ the age of tokens in places. A transition can be fired iff all tokens involved in the firing respect the residence interval in their places. Tokens are dropped with age 0. In strong semantics, if a token reaches its upper bound, and if there exists one firable transition that can consume this tokens, it must be fired.

For Place Time Petri Net, notations *enabled* and $firable$ are defined as follows:

$$t \in \mathit{enabled}(M\backslash \mathit{dead}) \text{ iff } M - \mathit{dead} \geq {}^\bullet t$$

$$t \in \mathit{firable}(M, \mathit{dead}, \nu) \text{ iff } \begin{cases} t \in \mathit{enabled}(M\backslash \mathit{dead}) \\ \forall p \in {}^\bullet t, \nu(p) \in I(p) \end{cases}$$

Definition 10 (Strong Semantics of *P-TPN*). *The semantics of a P-TPN $\mathcal{N}$ is a timed transition system $S_{\mathcal{N}} = (Q, q_0, \to)$ where: $Q = \{0,1\}^P \times \{0,1\}^P \times (\mathbb{R}_{\geq 0})^P$, $q_0 = (M_0, \mathbf{0}, \mathbf{0})$, $\longrightarrow \in Q \times (\Sigma_\varepsilon \cup \mathbb{R}_{\geq 0}) \times Q$ consists of the discrete and continuous transition relations:*
The discrete transition relation is defined $\forall t \in T$:

$$(M, dead, \nu) \xrightarrow{\Lambda(t)} (M', dead, \nu') \text{ iff } \begin{cases} t \in \mathit{firable}(M, dead, \nu) \\ M' = M - {}^\bullet t + t^\bullet \\ \nu'(p) = \begin{cases} 0 & \text{if } (dead(p) = 0) \wedge (p \in t^\bullet) \\ \nu(p) & \text{otherwise.} \end{cases} \end{cases}$$

The continuous transition relation is defined $\forall d \in \mathbb{R}_{\geq 0}$:

$$(M, dead, \nu) \xrightarrow{d} (M, dead', \nu') \text{ iff}$$

$$\begin{cases} \nu' = \nu + d \\ \forall t \in T : t \notin \mathit{firable}(M, dead, v + d) \Rightarrow (\forall d' \in [0, d] : t \notin \mathit{firable}(M, dead, v + d')) \\ dead'(p) = \begin{cases} 1 & \text{if } (p \in M \backslash dead) \wedge (v'(p) \notin I(p)^{\downarrow}) \\ dead(p) & \text{otherwise} \end{cases} \end{cases}$$

Definition 11 (Weak Semantics of *P-TPN*). *The weak semantics is exactly the same as the strong one without the condition $\forall t \in T : t \notin \mathit{firable}(M, dead, v + d) \Rightarrow (\forall d' \in [0, d] : t \notin \mathit{firable}(M, dead, v + d'))$ in the continuous transition relation.*

3.4 Arc Time Petri Nets (*A-TPN*)

The model. Arc Time Petri Nets were introduced in [23], adding interval on arcs and considering a weak semantics.

Like in *P-TPN*, an age is associated to each token. A transition t can be fired iff the tokens in the input places p satisfy the constraint on the arc from the place to the transition.

As for *P-TPN*, there could exist *dead tokens*, that is to say, tokens whose age is greater than the upper bound of all output arcs.

Definition 12 (Arc Time Petri Net). *An* Arc Time Petri Net *$\mathcal{N}$ is a tuple $(P, T, {}^\bullet(.), (.)^\bullet, M_0, I)$ where: $(P, T, {}^\bullet(.), (.)^\bullet, M_0)$ is a Petri net and $I : P \times T \to \mathcal{I}(\mathbb{Q}_{\geq 0})$ associates with each arc from place to transition a* time interval.

For Arc Time Petri Net, notations *enabled* and *firable* are defined as follows:

$$t \in \mathit{enabled}(M \backslash dead) \text{ iff } M - dead \geq {}^\bullet t$$

$$t \in \mathit{firable}(M, dead, \nu) \text{ iff } \begin{cases} t \in \mathit{enabled}(M \backslash dead) \\ \forall p \in {}^\bullet t, \nu(p) \in I(p, t) \end{cases}$$

Semantics of Arc Time Petri Nets. Like for *P-TPN*, the state of *A-TPN* is a tuple $(M, dead, \nu)$ where M is a marking, *dead* is the dead token mapping and $\nu \in \mathbb{R}^M_{\geq 0}$ the age of tokens in places. A transition t can be fired iff all tokens involved in the firing respect the constraint on arc from their place to the transition. Tokens are dropped with age 0. In strong semantics, if a token reaches one of its upper bound, and if there exists one transition that consumes this tokens, it must be fired.

Definition 13 (Strong Semantics of *A-TPN*). *The semantics of a P-TPN $\mathcal{N}$ is a timed transition system $S_{\mathcal{N}} = (Q, q_0, \to)$ where: $Q = \{0,1\}^P \times \{0,1\}^P \times (\mathbb{R}_{\geq 0})^P$, $q_0 = (M_0, \mathbf{0})$, $\longrightarrow \in Q \times (\Sigma_\varepsilon \cup \mathbb{R}_{\geq 0}) \times Q$ consists of the discrete and continuous transition relations: The discrete transition relation has the same definition that the one of A-TPN (with its specific definition of* firable*). The continuous transition relation is defined $\forall d \in \mathbb{R}_{\geq 0}$:*

$$(M, dead, \nu) \xrightarrow{d} (M, dead', \nu') \textit{ iff}$$

$$\begin{cases} \nu' = \nu + d \\ \forall t \in T : t \notin \mathit{firable}(M, dead, v + d) \Rightarrow (\forall d' \in [0, d] : t \notin \mathit{firable}(M, dead, v + d')) \\ dead'(p) = \begin{cases} 1 & \textit{if} \begin{cases} p \in M \backslash dead \\ \forall t \in p^\bullet, \nu'(p) \notin I(p,t)^\downarrow \end{cases} \\ dead(p) & \textit{otherwise} \end{cases} \end{cases} \tag{4}$$

The definition of semantics of *A-TPN* and *P-TPN* are very similar: the only difference is that, in the definition of *A-TPN*, the timing condition for firable is $\forall p \in {}^\bullet t : \nu(p) \in I(p,t)$ as in *P-TPN*, it's $\forall p \in {}^\bullet t : \nu(p) \in I(p)$, and the same for the condition associated to *dead*.

Definition 14 (Weak Semantics of *A-TPN*). *The weak semantics is exactly the same as the strong one without the condition $\forall t \in T : t \notin \mathit{firable}(M, dead, v + d) \Rightarrow (\forall d' \in [0, d] : t \notin \mathit{firable}(M, dead, v + d'))$ in the continuous transition relation.*

4 Comparison of the Expressiveness Wrt Bisimulation

In the sequel we will compare various classes of safe TPN w.r.t. bisimulation. We note $\overline{T\text{-}TPN}$ and $\underline{T\text{-}TPN}$, for the classes of safe Transition Time Petri nets respectively with strong and weak semantics. We note $\overline{A\text{-}TPN}$ and $\underline{A\text{-}TPN}$, for the classes of safe Arc Time Petri nets respectively with strong and weak semantics. We note $\overline{P\text{-}TPN}$ and $\underline{P\text{-}TPN}$, for the classes of safe Place Time Petri nets respectively with strong and weak semantics.

A *run* of a time Petri net $\mathcal{N}$ is a (finite or infinite) path in $S_{\mathcal{N}}$ starting in q_0. As a shorthand we write $s \xrightarrow{(t,d)} s'$ (where a state s is equal to (M, ν) or $(M, dead, \nu)$) for a sequence of time elapsing and discrete steps like $s \xrightarrow{d} s'' \xrightarrow{t} s'$. Moreover we write $\mathcal{N}$ for $S_{\mathcal{N}}$ (i.e. we will use the shorthand : *a run ρ of $\mathcal{N}$* or *a state s of $\mathcal{N}$*).

4.1 $\overline{X\text{-}TPN} \not\subseteq_{\approx} \underline{X\text{-}TPN}$ with $X \in \{T, A, P\}$

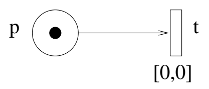


Fig. 1. A "non-delay" $\overline{T\text{-}TPN}$

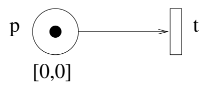


Fig. 2. A "non-delay" $\overline{P\text{-}TPN}$

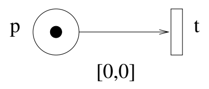


Fig. 3. A "non-delay" $\overline{A\text{-}TPN}$

Theorem 1 (Weak semantics can not emulate strong semantics)

$$\overline{P\text{-}TPN} \not\subseteq_{\approx} \underline{P\text{-}TPN} \qquad \overline{T\text{-}TPN} \not\subseteq_{\approx} \underline{T\text{-}TPN} \qquad \overline{A\text{-}TPN} \not\subseteq_{\approx} \underline{A\text{-}TPN}$$

Proof. By contradiction: assume it exist a $\underline{T\text{-}TPN}$ weakly timely bisimular to the $\overline{T\text{-}TPN}$ of Figure 1. From its initial state, a delay of duration $d > 0$ is possible (in weak semantics, a delay is always possible). By bisimulation hypothesis, it should also be possible from the initial state of the strong $T\text{-}TPN$ of Figure 1. This contradicts our assumption.

The same applies for $P\text{-}TPN$ and $A\text{-}TPN$. □

4.2 $\underline{P\text{-}TPN} \subset_{\approx} \overline{P\text{-}TPN}$

Let $\mathcal{N} \in \underline{P\text{-}TPN}$. We construct a TPN $\overline{\mathcal{N}} \in \overline{P\text{-}TPN}$ as follow :

- we start from $\overline{\mathcal{N}} = \mathcal{N}$ and $\overline{M_0} = M_0$,
- for each place p of $\mathcal{N}$,
  - we add in $\overline{\mathcal{N}}$, the net in the gray box of the figure 4 with a token in place p_2^t.
  - for each transition t such that $p \in {}^\bullet t$, we add an arc from p_2^t to t and an arc from t to p_2^t.

Note that in the gray box, there is always a token either in place p_1^t or in the place p_2^t.

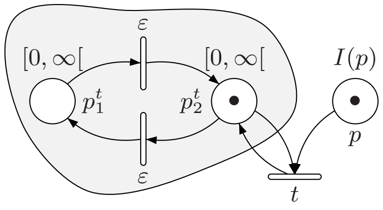


Fig. 4. The translation from $\underline{P\text{-}TPN}$ into $\overline{P\text{-}TPN}$

Lemma 1 (Translating a $\underline{P\text{-}TPN}$ into a $\overline{P\text{-}TPN}$). *Let $\mathcal{N} \in \underline{P\text{-}TPN}$ and $\overline{\mathcal{N}} \in \overline{P\text{-}TPN}$ its translation into $\overline{P\text{-}TPN}$ as defined previously, $\mathcal{N}$ and $\overline{\mathcal{N}}$ are timed bisimilar.*

Proof. $\mathcal{N} = (P, T, {}^\bullet(.), (.)^\bullet, M_0, I)$ and $\overline{\mathcal{N}} = (\overline{P}, \overline{T}, \overline{{}^\bullet(.)}, \overline{(.)^\bullet}, \overline{M_0}, \overline{I})$. Note that $P \subset \overline{P}$ and $T \subset \overline{T}$.

Let $(M, dead, \nu)$ be a state of $\mathcal{N}$ and $(\overline{M}, \overline{dead}, \overline{\nu})$ be a state of $\overline{\mathcal{N}}$. We define the relation $\approx \subseteq ((\{0,1\} \times \mathbb{R}_{\geq 0})^P \times (\{0,1\} \times \mathbb{R}_{\geq 0})^{\overline{P}}$ by:

$$(M, dead, \nu) \approx (\overline{M}, \overline{dead}, \overline{\nu}) \iff \forall p \in P \begin{cases} (1)\, M(p) = \overline{M}(p) \\ (2)\, dead(p) = \overline{dead}(p) \\ (3)\, \nu(p) = \overline{\nu}(p) \end{cases} \tag{5}$$

Now we can prove that $\approx$ is a weak timed bisimulation relation between $\mathcal{N}$ and $\overline{\mathcal{N}}$.

Proof : First we have $(M_0, dead_0, \nu_0) \approx (\overline{M_0}, \overline{dead_0}, \overline{\nu_0})$.

Let us consider a state $s = (M, dead, \nu) \in \mathcal{N}$ and a state $\overline{s} = (\overline{M}, \overline{dead}, \overline{\nu}) \in \overline{\mathcal{N}}$ such that $(\overline{M}, \overline{dead}, \overline{\nu}) \approx (M, dead, \nu)$.

- *Discrete transitions* Let t be a firable transition from $s = (M, dead, \nu)$ in $\mathcal{N}$. There is a run $\rho_1 = (M, dead, \nu) \xrightarrow{t} (M_1, dead_1, \nu_1)$ (with $dead = dead_1$). It means that $\forall p \in {}^\bullet(t)\ \nu(p) \in I(p)^\downarrow$. Moreover, $M_1 = M - {}^\bullet t + t^\bullet$ and $\forall p \in M_1 \backslash dead_1,\ \nu_1(p) = 0$ if $p \in t^\bullet$.
  In $\overline{\mathcal{N}}$, as $(\overline{M}, \overline{dead}, \overline{\nu}) \approx (M, dead, \nu)$ we have $\forall p \in {}^\bullet(t)\ \overline{\nu}(p) \in \overline{I}(p)^\downarrow$. Moreover $\overline{\nu}(p_2^t) \in \overline{I}(p_2^t)^\downarrow$(with upper bound : ∞) and there is a token either in p_1^t or in p_2^t. Thus, there is a run $\overline{\rho}_1 = (\overline{M}, \overline{dead}, \overline{\nu}) \xrightarrow{\epsilon*} (\overline{M}_{\varepsilon_1}, \overline{dead}_{\varepsilon_1}, \overline{\nu}_{\varepsilon_1}) \xrightarrow{t} (\overline{M}_1, \overline{dead}_1, \overline{\nu}_1)$ with $(\overline{M}_{\varepsilon_1}, \overline{dead}_{\varepsilon_1}, \overline{\nu}_{\varepsilon_1}) \approx (M, dead, \nu)$ and $\overline{M}_{\varepsilon_1}(p_2^t) = 1$. We have $\overline{M_1} = \overline{M} - \overline{{}^\bullet t} + \overline{t^\bullet}$ that is to say $\overline{M}_1(p_2^t) = 1$, $\overline{M}_1(p_1^t) = 0$ and $\forall p \in P$, $\overline{M}_1(p) = M_1(p)$. Moreover $\overline{dead} = \overline{dead_1}$ and $\forall p \in \overline{M}_1 \backslash \overline{dead}_1, \overline{\nu}_1(p) = 0$ if $p \in \overline{t^\bullet}$ and then $\forall p \in P, \overline{\nu}_1(p) = \nu_1(p)$. Thus $(\overline{M}_1, \overline{dead}_1, \overline{\nu}_1) \approx (M_1, dead_1, \nu_1)$.
- *Continuous transitions* In $\mathcal{N}$, from $s = (M, dead, \nu)$, there is a run $\rho_2 = (M, dead, \nu) \xrightarrow{d} (M_2, dead_2, \nu_2)$ such that $\forall p \in M(p), \nu_2(p) = \nu(p) + d$ and $M = M_2$. Moreover, $\forall p \in M \backslash dead$, $M_2(p) = 1$ and $dead_2(p) = 0$ if $\nu_2(p) \in I(p)^\downarrow$ and $M_2(p) = dead_2(p) = 1$ if $\nu_2(p) \notin I(p)^\downarrow$.
  - if there is no firable transition t such that $\exists p_t \in {}^\bullet(t)$ with $\nu(p_t) \in I(p_t)^\downarrow$ and $\nu_2(p_t) \notin I(p_t)^\downarrow$. As $(M, dead, \nu) \approx (\overline{M}, \overline{dead}, \overline{\nu})$, we have $\forall p \in P$, $M(p) = \overline{M}(p)$, $dead(p) = \overline{dead}(p)$ and $\nu(p) = \overline{\nu}(p)$ and then in $\overline{\mathcal{N}}$, there is a run $\overline{\rho}_2 = (\overline{M}, \overline{dead}, \overline{\nu}) \xrightarrow{d} (\overline{M}_2, \overline{dead}_2, \overline{\nu}_2)$ such that $\overline{dead}_2 = \overline{dead}$ and $\forall p \in P$, $\overline{M}_2(p) = M_2(p)$ and $\overline{\nu}_2(p) = \overline{\nu}_2(p) + d = \nu_2(p)$. Thus $(\overline{M}_2, \overline{dead}_2, \overline{\nu}_2) \approx (M_2, dead_2, \nu_2)$
  - if there is a firable transition t such that $\exists p_t \in {}^\bullet(t)$ with $\nu(p_t) \in I(p_t)^\downarrow$ and $\nu_2(p_t) \notin I(p_t)^\downarrow$ (and then $dead(p_t) = 0$ and $dead_2(p_t) = 1$). As $(M, dead, \nu) \approx (\overline{M}, \overline{dead}, \overline{\nu})$, we have $\forall p \in P$, $M(p) = \overline{M}(p)$, $dead(p) = \overline{dead}(p)$ and $\nu(p) = \overline{\nu}(p)$. In $\overline{\mathcal{N}}$, there is a run $\overline{\rho}_2 = (\overline{M}, \overline{dead}, \overline{\nu}) \xrightarrow{\epsilon*} (\overline{M}_{\varepsilon_2}, \overline{dead}_{\varepsilon_2}, \overline{\nu}_{\varepsilon_2})$ such that $(\overline{M}_{\varepsilon_2}, \overline{dead}_{\varepsilon_2}, \overline{\nu}_{\varepsilon_2}) \approx (M, dead, \nu)$ and $\overline{M}_{\varepsilon_2}(p_2^t) = 0$. Thus, there is a run $(\overline{M}_{\varepsilon_2}, \overline{dead}_{\varepsilon_2}, \overline{\nu}_{\varepsilon_2}) \xrightarrow{d} (\overline{M}_2, \overline{dead}_2, \overline{\nu}_2)$ such that $\overline{M}_2(p_t) = \overline{dead}_2(p_t) = 1$ and $\forall p \in P$, $\overline{\nu_2}(p) = \overline{\nu}(p) + d = \nu_2(p)$ and then $(\overline{M}_2, \overline{dead}_2, \overline{\nu}_2) \approx (M_2, dead_2, \nu_2)$.

The converse is straightforward following the same steps as the previous ones. □

Theorem 2. $\underline{P\text{-}TPN} \subset_{\approx} \overline{P\text{-}TPN}$

Proof. As $\overline{P\text{-}TPN} \not\subseteq_{\approx} \underline{P\text{-}TPN}$ and thanks to Lemma 1. □

4.3 $\underline{A\text{-}TPN} \subset_{\approx} \overline{A\text{-}TPN}$

Theorem 3. $\underline{A\text{-}TPN} \subset_{\approx} \overline{A\text{-}TPN}$

Proof. As for Theorem 2 □

4.4 $\underline{T\text{-}TPN} \not\subseteq_{\approx} \overline{T\text{-}TPN}$

We first recall the following theorem :

Theorem 4 ([3]). *There is no TPN* $\in \overline{T\text{-}TPN}$ *weakly timed bisimilar to* $\mathcal{A}_0 \in \mathcal{TA}$ *(Fig. 5).*

Theorem 5. $\underline{T\text{-}TPN} \not\subseteq_{\approx} \overline{T\text{-}TPN}$

Proof. We first prove that the TPN $\mathcal{N}_{T0} \in \underline{T\text{-}TPN}$ of Fig. 6 is weakly timed bisimilar to $\mathcal{A}_0 \in \mathcal{TA}$ (Fig. 5).
Let (ℓ, v) be a state of $\mathcal{A}_0 \in \mathcal{TA}$ where $\ell \in \{\ell_0, \ell_1\}$ and $v(x) \in \mathbb{R}_{\geq 0}$ is the valuation of the clock x. We define the relation $\approx \subseteq (\{\ell_0, \ell_1\} \times \mathbb{R}_{\geq 0}) \times (\{0,1\} \times \mathbb{R}_{\geq 0})$ by:

$$(\ell, v) \approx (M, \nu) \iff \begin{cases} (1)\, \ell = \ell_0 \iff M(P_1) = 1 \\ \quad\;\; \ell = \ell_1 \iff M(P_1) = 0 \\ (2)\, v(x) = \nu(a) \end{cases} \quad (6)$$

$\approx$ is a weak timed bisimulation (The proof is straightforward).
From Theorem 4, there is no TPN $\in \overline{T\text{-}TPN}$ weakly timed bisimilar to $\mathcal{A}_0 \in \mathcal{TA}$ (Fig. 5) and the TPN $\mathcal{N}_{T0} \in \overline{T\text{-}TPN}$ of Fig. 6 is weakly timed bisimilar to $\mathcal{A}_0$. □

4.5 $\overline{P\text{-}TPN} \not\subseteq_{\approx} \overline{T\text{-}TPN}$

Lemma 2. *The TPN* $\mathcal{N}_{P0} \in \overline{P\text{-}TPN}$ *(Fig.7) is weakly timed bisimilar to* $\mathcal{A}_0 \in \mathcal{TA}$ *(Fig. 5).*

Proof. From Lemma 1, $\mathcal{N}_{P0} \approx \mathcal{N}_{P1}$. Obviously, $\mathcal{N}_{P1} \approx \mathcal{N}_{T0}$. And, from proof of Theorem 5, $\mathcal{N}_{T0} \approx \mathcal{A}_0$. By transitivity, $\mathcal{N}_{P0} \approx \mathcal{A}_0$.
($\mathcal{N}_{P0}$, $\mathcal{N}_{P1}$, $\mathcal{N}_{T0}$ and $\mathcal{A}_0$ are respectivly presented in Figures 7, 8, 6, 5). □

Theorem 6

$$\overline{P\text{-}TPN} \not\subseteq_{\approx} \overline{T\text{-}TPN}$$

Proof. From Theorem 4, there is no TPN $\in \overline{T\text{-}TPN}$ weakly timed bisimilar to $\mathcal{A}_0 \in \mathcal{TA}$ (Fig. 5) and the TPN $\mathcal{N}_{P0} \in \overline{P\text{-}TPN}$ is weakly timed bisimilar to $\mathcal{A}_0$. □

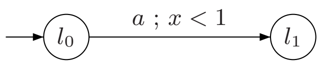


Fig. 5. The Timed Automaton $\mathcal{A}_0$

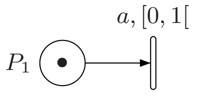


Fig. 6. The TPN $\mathcal{N}_{T0} \in \underline{T\text{-}TPN}$ bisimilar to $\mathcal{A}_0$

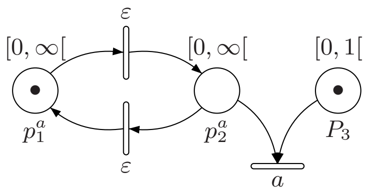


Fig. 7. The TPN $\mathcal{N}_{P0} \in \overline{P\text{-}TPN}$ bisimilar to $\mathcal{A}_0$

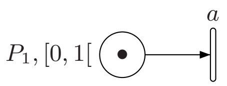


Fig. 8. A TPN $\mathcal{N}_{P1} \in \underline{P\text{-}TPN}$ bisimilar to $\mathcal{N}_{P0}$

4.6 $\overline{\textit{T-TPN}} \not\subseteq_{\approx} \overline{\textit{P-TPN}}$ and $\underline{\textit{T-TPN}} \not\subseteq_{\approx} \underline{\textit{P-TPN}}$

Definition 15 (Relevant clock of a *P-TPN*). *Let $\mathcal{N} = (P, T, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$ be a P-TPN ($\overline{P\text{-}TPN}$ or $\underline{P\text{-}TPN}$), and $s = (M, dead, \nu)$ be a state of $\mathcal{N}$. In s, a clock x associated to a place $p \in P$ is said to be* relevant *iff $M(p) = 1$.*

We first give a lemma stating that "in *P-TPN* ($\overline{P\text{-}TPN}$ or $\underline{P\text{-}TPN}$) a relevant clock (associated to a token in a marked place p) can become irrelevant or can be reset only in its firing interval ($\nu(p) \in I(p)$) ".

Lemma 3 (Reset of relevant clock in *P-TPN*). In *P-TPN*, a relevant clock can become irrelevant or can be reset only in its firing interval. *Let $\mathcal{N}$, be a P-TPN ($\overline{P\text{-}TPN}$ or $\underline{P\text{-}TPN}$). Let $(M, dead, \nu)$ be a state of $\mathcal{N}$ such that $M(p) > 0$ and $\nu(p) > 0$. If $(M, dead, \nu) \rightarrow (M', dead', \nu')$ (where $\rightarrow$ is a discrete or a continuous transition) and $\nu'(p) = 0$ or $M'(p) = 0$ then $\nu(p) \in I(p)$*

Proof. From the semantics of *P-TPN* ($\overline{P\text{-}TPN}$ or $\underline{P\text{-}TPN}$), a relevant clock associated to a place p ($M(p) = 1$) can become irrelevant or can be reset only by a discrete transition $(M, dead, \nu) \xrightarrow{t} (M', dead, \nu')$ such that $p \in {}^\bullet t$ (if $p \in t^\bullet$ the relevant clock is reset, otherwise it become irrelevant). Then, as $t \in \textit{firable}(M, dead, \nu)$, we have $\nu(p) \in I(p)$. □

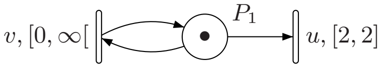


Fig. 9. The TPN $\mathcal{N}_{T1} \in \overline{T\text{-}TPN}$

Theorem 7. *There is no TPN $\in \overline{P\text{-}TPN}$ weakly timed bisimilar to $\mathcal{N}_{T1} \in \overline{T\text{-}TPN}$ (Fig. 9).*

Proof. The idea of the proof is that in the *T-TPN* $\mathcal{N}_{T1}$ the clock associated to the transition u can be reset at any time (in particular before 2 time units). In the *P-TPN*, time measure is performed by a finite number of clock. We are going to do this reset more times than this number, outside of their firing interval, leading to contradiction of Lemma 3.

Assume there is a *P-TPN* $\mathcal{N}' = (P', T', {}^\bullet(.)', (.)'^\bullet, M'_0, \Lambda', I')$ that is weakly timed bisimilar to $\mathcal{N}_{T1}$. Let us define $P_> \subseteq P'$, the finite set of n places p of $\mathcal{N}'$ such that the lower bound $\alpha'(p)$ (see section 2 for notations α and β) of interval $I'(p)$ is non nul. Let us define $\delta_\alpha = \min\{\alpha'(p) \,|\, p \in P_>\}$, $\delta_\beta = \min\{\beta'(p) \neq 0 \,|\, p \in P'\}$, and $\delta = \min \delta_\alpha, \delta_\beta$.

Let $s_0 = (M_0, \nu_0)$ be the initial state of $\mathcal{N}_{T1}$ and $s'_0 = (M'_0, dead'_0, \nu'_0)$ the initial state of $\mathcal{N}'$.

The proof is decomposed into three steps:

1. *$P_>$ is non empty:*

   Proof. Let ρ be a run of $\mathcal{N}_{T1}$, $\rho = s_0 \xrightarrow{2} s_2 \xrightarrow{u} s_u$.

   In $\mathcal{N}'$, $\exists \rho' = s'_0 \xrightarrow{(\epsilon*,2)} s'_2 \xrightarrow{(\epsilon*,u)} s'_u$, such that s_0, s_2 and s_u are respectively bisimilar to s'_0, s'_2 and s'_u.

   If $P_>$ is empty, the sequence $s'_0 \xrightarrow{(\epsilon*,0)}\xrightarrow{(\epsilon*,u)}$ exists, and then, u is fired at date 0, which is a contradiction. □

   This means that $P_>$ is non empty, δ exists and a subset of $P_>$ is used to measure the 2 time units elapsing from initial state up to s_2.

2. Let us consider $\rho_\tau = s_0 \xrightarrow{\tau} s_\tau \xrightarrow{2-\tau} s_2 \xrightarrow{u} s_3$ in $\mathcal{N}_{T1}$ and $\rho'_\tau = s'_0 \xrightarrow{(\varepsilon*,\tau)} s'_\tau \xrightarrow{(\varepsilon*,2-\tau)} s'_2 \xrightarrow{u} s'_3$ its equivalent in $\mathcal{N}'$.

   We will now prove that, for all $\tau < \delta$, it exists $p \in P'$ such that, in s'_τ, $M'(p) = 1$ and $\nu'(p) = \tau$.

   Proof. First, notice that all marked places are consistent ones, because $\tau < \delta_\beta$.

   Assume that there is no consistent clock with value τ in s'_τ.

   Each consistent clock whose value is $\tau' < \tau$ in s'_τ has been enabled at time $\tau - \tau'$. Since $\tau - \tau' < \delta$, the same run from s'_0 to $s_{\tau-\tau'}$ can be done in 0 time. Consequently, the state s'_τ can be reached by a run of duration $\tau' < \tau$, which contradict the bisimilation relation between s_τ and s'_τ.

   Then for all $\tau < \delta$, it exists $p \in P'$ such that, in s'_τ, $M'(p) = 1$ and $\nu'(p) = \tau$. Moreover, thanks to item 1, $p \in P_>$. □

3. In $\mathcal{N}_{T1}$, from s_τ, the firing of v leads to a state bisimilar to s_0 and then in $\mathcal{N}'$, from s'_τ, the firing of v leads to a state s^1_0 bisimilar to s_0.

   Let us consider the run $s_0 \xrightarrow{\tau_1} s_{\tau_1} \xrightarrow{v} s_0 \xrightarrow{\tau_2} s_{\tau_2} \xrightarrow{v} s_0 \cdots \xrightarrow{\tau_k} s_{\tau_k} \xrightarrow{v} s_0$ in $\mathcal{N}_{T1}$ with all $\tau_i > 0$ and $\rho'' = s'_0 \xrightarrow{(\varepsilon*,\tau_1)} s'_{\tau_1} \xrightarrow{v} s^1_0 \xrightarrow{(\varepsilon*,\tau_2)} s'_{\tau_2} \xrightarrow{v} s^2_0 \cdots \xrightarrow{(\varepsilon*,\tau_k)} s'_{\tau_k} \xrightarrow{v} s^k_0$ its equivalent in $\mathcal{N}'$.

Each state s_0^i is bisimilar to s_0 and then, for each s_0^i there is a relevant clock associated to a place $p \in P_>$ whose value is equal to zero (application of previous item with $\tau = 0$). Now, assume $k > n$ (n is the size of $P_>$) and $\sum_{1 \leq i \leq n} \tau_i < \delta$, at least one relevant clock (associated to a place $p \in P_>$) has to become irrelevant or has to reset in the run ρ'' whereas $\nu_\tau(p) \notin I(p)$ contradicting the Lemma 3. □

Corollary 1

$$\overline{T\text{-}TPN} \not\subseteq_\approx \overline{P\text{-}TPN}$$

Proof. Direct from Theorem 7. □

Moreover, the Theorem 7 remains valid in weak semantics. Indeed, we can consider the net of the Fig. 9 with a weak semantic and the proof of Theorem 7 remains identical. We have then the following corollary.

Corollary 2

$$\underline{T\text{-}TPN} \not\subseteq_\approx \underline{P\text{-}TPN}$$

4.7 $\overline{T\text{-}TPN} \subset_\approx \overline{A\text{-}TPN}$ and $\underline{T\text{-}TPN} \subseteq_\approx \underline{A\text{-}TPN}$

The proof of this strict inclusion is done in two steps: Lemma 4 (in Section 4.7) shows that $\overline{T\text{-}TPN} \subseteq_\approx \overline{A\text{-}TPN}$ (by construction: for each $\overline{T\text{-}TPN}$, a weak-bisimilar $\overline{A\text{-}TPN}$ is built), and Lemma 5 shows that it exists a $\overline{A\text{-}TPN}$ bisimilar to $\mathcal{A}_0 \in \mathcal{TA}$ (Fig. 5) already used in Theorem 4. With these two lemmas, the strict inclusion is straightforward (Section 4.7).

Weak Inclusion: : $\overline{T\text{-}TPN} \subseteq_\approx \overline{A\text{-}TPN}$ and $\underline{T\text{-}TPN} \subseteq_\approx \underline{A\text{-}TPN}$

Lemma 4 (From *T-TPN* to *A-TPN*)

$$\overline{T\text{-}TPN} \subseteq_\approx \overline{A\text{-}TPN} \qquad\qquad \underline{T\text{-}TPN} \subseteq_\approx \underline{A\text{-}TPN}$$

The proof is done by construction: for each $T\text{-}TPN \mathcal{N}$, a weak-bisimilar $A\text{-}TPN \mathcal{N}'$ is built. The main issue is to emulate the $T\text{-}TPN$ "start clock when all input places are marked" rule with the $A\text{-}TPN$ rule "start clock as soon as the token is in place".

The main idea is, for each transition t in a $T\text{-}TPN$ $\mathcal{N}$, a *chain* of places $^\circ t^0, \ldots, {}^\circ t^n$ (with $n = |{}^\bullet t|$) is built in the translated $A\text{-}TPN$ $\mathcal{N}'$, such that $\sum_{p \in {}^\bullet t} M_\mathcal{N}(p) = i \iff M_{\mathcal{N}'}({}^\circ t^i) = 1$ (with $i \in [1, n]$). Therefor, the time interval $I_\mathcal{N}(t)$ is set to arc from ${}^\circ t^{|{}^\bullet t|}$ to t. Then, the rule "start clock in $I(t)$ when all input places of t are marked" is emulated by the rule "start clock constraint in $I({}^\circ t^{|{}^\bullet t|}, t)$ when ${}^\circ t^{|{}^\bullet t|}$ is marked" which is equivalent because $I_{\mathcal{N}'}({}^\circ t^{|{}^\bullet t|}, t) = I_\mathcal{N}(t)$ and $\sum_{p \in {}^\bullet t} M_\mathcal{N}(p) = n \iff M_{\mathcal{N}'}({}^\circ t^n) = 1$.

Of course, the transitions that modify ${}^\bullet t$ in $\mathcal{N}$ should have a matching transition in $\mathcal{N}$ that modifies ${}^\circ t^i$.

Example. The *T-TPN* of Figure 10 is translated into the *A-TPN* of Figure 11. Then, the firing condition associated to t is activated only when there is one token in place $^\circ t^{|^\bullet t|}$ ($^\circ t^2$ in the example), that is to say, when there are enough tokens in the emulated places $^\bullet t$.

With this chain structure, the firing of the transition u (resp. v) must increase the marking of $^\bullet t$, *i.e.* put a token in $^\circ t^1$ *or* $^\circ t^2$ (depending on the previous marking). To achieve this goal, since bisimulation is based on the timed transition system where only labels of transitions are visible, the transition u can be replaced by two transitions, one putting a token in $^\circ t^1$ and the other in $^\circ t^2$, as long as they have the same label. In Figure 11, these two transitions are called $u_{(t:0,1)}$ and $u_{(t:1,2)}$ and $\Lambda(u_{(t:0,1)}) = \Lambda(u_{(t:1,2)}) = \Lambda(u)$ ([6]).

Once this done, a little stuff has to be added to handle conflict and reversible nets[7]. It should be noticed that the exactly the same translation applies for weak and strong semantics.

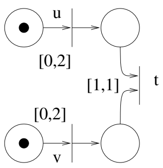


Fig. 10. A *T-TPN*

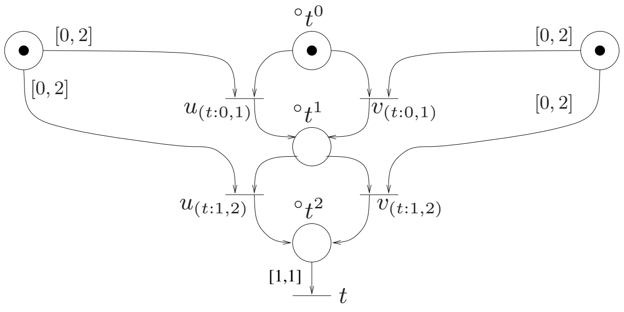


Fig. 11. A translation of the *T-TPN* of Figure 10 into *A-TPN*

By lack of space, the details of the translation and the proof are not presented here and can be found in [9].

A Specific $\overline{A\text{-}TPN}$

Lemma 5. *The TPN $\mathcal{N}_{A0} \in \overline{A\text{-}TPN}$ of Fig.13 is weakly timed bisimilar to $\mathcal{A}_0 \in \mathcal{TA}$ (Fig. 5).*

The bisimulation relation and the proof are identical to those of Lemma 2.

Strict Inclusion in Strong Semantics

Theorem 8

$$\overline{T\text{-}TPN} \subset_\approx \overline{A\text{-}TPN}$$

[6] Notation $u_{(t:1,2)}$ is used to denotes that this firing of u makes the marking of $^\bullet t$ going form 1 to 2.

[7] This translation pattern have been used in [7] to translate $\overline{T\text{-}TPN}$ into $\overline{P\text{-}TPN}$, but it was a mistake. The translation only apply in some specific cases: when transitions are conflict-free or when the lower bound of time intervals is 0 for example (see[9]).

Proof. Thanks to Lemma 4 we have $\overline{T\text{-}TPN} \subseteq_{\approx} \overline{A\text{-}TPN}$. Moreover from Theorem 4, there is no TPN $\in \overline{T\text{-}TPN}$ weakly timed bisimilar to $\mathcal{A}_0 \in \mathcal{TA}$ (Fig. 5) and from Lemma 5, the TPN $\mathcal{N}_{A0} \in \overline{A\text{-}TPN}$ is weakly timed bisimilar to $\mathcal{A}_0$. □

4.8 $\overline{P\text{-}TPN} \subset_{\approx} \overline{A\text{-}TPN}$ and $\underline{P\text{-}TPN} \subset_{\approx} \underline{A\text{-}TPN}$

Lemma 6 **(*P-TPN* included in *A-TPN* (strong and weak semantics))**

$$\underline{P\text{-}TPN} \subseteq_{\approx} \underline{A\text{-}TPN} \qquad\qquad \overline{P\text{-}TPN} \subseteq_{\approx} \overline{A\text{-}TPN}$$

Proof. The translation is obvious: for a given *P-TPN* $\mathcal{N}$, a *A-TPN* $\mathcal{N}'$ is built, with the same untimed Petri net, and such that, $\forall p, \forall t \in p^{\bullet} : I'(p,t) = I(p)$. Then, considering their respective definitions for *enabled*, $firable$ and the discrete and continuous translation, the only difference is that, when the *P-TPN* condition is $\nu(p) \in I(p)$ or $\nu(p) \in I(p)^{\downarrow}$, the *A-TPN* condition is $\forall t \in p^{\bullet} : \nu(p) \in I(p,t)$ or $\nu(p) \in I(p,t)^{\downarrow}$. And in our translation, $I'(p,t) = I(p)$.

Then, all evolution rules are the same and both are strongly bisimilar. □

Lemma 7 **(No $\overline{P\text{-}TPN}$ is bisimilar to a $\overline{A\text{-}TPN}$).** *It exists $\mathcal{N}_{A1} \in \overline{A\text{-}TPN}$ such that there is no $\mathcal{N} \in \overline{P\text{-}TPN}$ weakly timed bisimilar to $\mathcal{N}_{A1}$.*

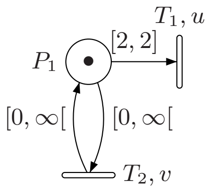


Fig. 12. The TPN $\mathcal{N}_{A1} \in \overline{A\text{-}TPN}$

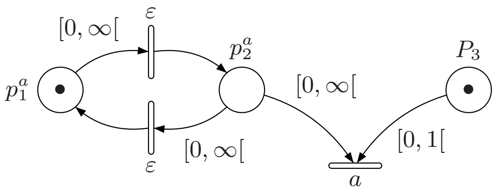


Fig. 13. The TPN $\mathcal{N}_{A0} \in \overline{A\text{-}TPN}$ bisimilar to $\mathcal{A}_0$

Proof. The proof is based on Theorem 7. The *A-TPN* $\mathcal{N}_{A1}$ (cf. Fig. 12) is the same net than the *T-TPN* $\mathcal{N}_{T1}$ (cf. Fig. 9)). Obviously, $\mathcal{N}_{A1}$ and $\mathcal{N}_{T1}$ are (strongly) bisimilar. Then, from Theorem 7 that states that there is no $\overline{P\text{-}TPN}$ weakly bisimilar to $\mathcal{N}_{T1}$, there neither is any $\overline{P\text{-}TPN}$ weakly bisimilar to $\mathcal{N}_{A1}$. □

Lemma 8 **(No $\underline{P\text{-}TPN}$ is bisimilar to a $\underline{A\text{-}TPN}$).** *It exists $\mathcal{N}_{A1} \in \underline{A\text{-}TPN}$ such that there is no $\mathcal{N} \in \underline{P\text{-}TPN}$ weakly timed bisimilar to $\mathcal{N}_{A1}$.*

The proof is the same as for Lemma 7.

Theorem 9 **(*A-TPN* are strictly more expressive than *P-TPN*).**

$$\overline{P\text{-}TPN} \subset_{\approx} \overline{A\text{-}TPN} \qquad\qquad \underline{P\text{-}TPN} \subset_{\approx} \underline{A\text{-}TPN}$$

Proof. Obvious from Lemma 6, 7 and 8.

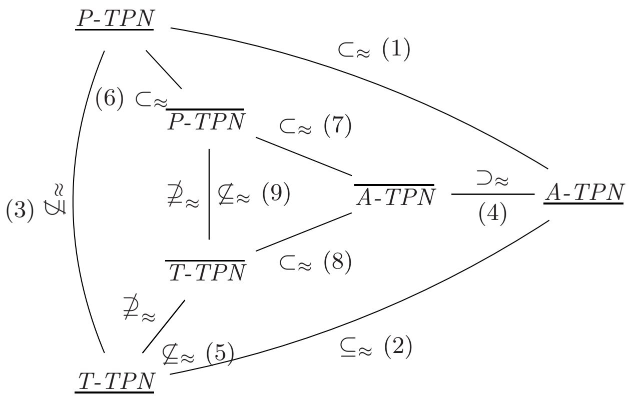

$\underline{P\text{-}TPN}$
$\subset_\approx$ (1)
(6) $\subset_\approx$
$\overline{P\text{-}TPN}$
$\subset_\approx$ (7)
(3) $\not\subseteq_\approx$
$\not\supseteq_\approx$
$\not\subseteq_\approx$ (9)
$\overline{A\text{-}TPN}$
$\supset_\approx$
(4)
$\underline{A\text{-}TPN}$
$\overline{T\text{-}TPN}$
$\subset_\approx$ (8)
$\not\supseteq_\approx$
$\not\subseteq_\approx$ (5)
$\subseteq_\approx$ (2)
$\underline{T\text{-}TPN}$

Fig. 14. The classification explained

4.9 Sum Up

We are now going to sum-up all results in a single location, Figure 14.

(1) and (7) A *P-TPN* can always be translated into a *A-TPN* and there exist some *A-TPN* that can not be simulated by any *P-TPN* (Theorem 9).

(2) A $\underline{T\text{-}TPN}$ can be translated into a $\underline{A\text{-}TPN}$ (Lemma 4). Then, $\underline{A\text{-}TPN}$ are more expressive than $\underline{T\text{-}TPN}$. Is this relation strict or not is still an open problem.

(3) Corrolary 2 states that $\underline{T\text{-}TPN} \not\subseteq_\approx \underline{P\text{-}TPN}$ But we do not know more: does it mean that $\underline{P\text{-}TPN}$ are more expressive than $\underline{T\text{-}TPN}$, or are both models incomparable is still another open problem.

(4) The strong semantics of *A-TPN* strictly generalise the weak one (Theorem 3).

(5) Strong and weak *T-TPN* are incomparable: the weak semantics can not emulate the strong one (Theorem 1) but there also exist *T-TPN* with weak semantics that can not been emulated by any strong *T-TPN* (Theorem 4).

(6) Theorem 2 states that $\underline{P\text{-}TPN} \subset_\approx \overline{P\text{-}TPN}$: in *P-TPN*, the strong semantics can emulate the weak one (Lemma 1), but weak semantic can not do the opposite (Theorem 1).

(8) A $\overline{T\text{-}TPN}$ can be translated into a $\overline{A\text{-}TPN}$ (Lemma 4) and there exists a $\overline{A\text{-}TPN}$ (Lemma 5) that can not be emulated by any $\overline{T\text{-}TPN}$. Then strict inclusion follows (Theorem 8).

(9) *T-TPN* and *P-TPN* with strong semantics are incomparable: Theorem 6 states that there is a $\overline{P\text{-}TPN}$ that can be simulated by no $\overline{T\text{-}TPN}$ and Corollary 1 states the symmetric.

5 Conclusion

Several timed Petri nets models have been defined for years and different purposes. They have been individually studied, some analysis tools exist for some, and the users know that a given problem can be modelled with one or the other with more or less difficulty, but a clear map of their relationships was missing. This paper draws most of this map (cf. Fig. 14).

Behind the details of the results, a global view of the main results is following:

– *P-TPN* and *A-TPN* are really close models, since their firing rule is the conjunction of some local clocks, whereas the *T-TPN* has another point of view, its firing rule taking into account only the last clock;
– the *A-TPN* model generalises all the other models, but emulating the *T-TPN* firing rule with *A-TPN* ones is not possible in practice for human modeller;
– the strong semantics generalise the weak one for *P-TPN* and *A-TPN*, but not for *T-TPN*.

There are still two open problems related to the weak semantics of *T-TPN*: "is the inclusion of *T-TPN* into *A-TPN* strict?" and "does *T-TPN* generalise *P-TPN* or are they incomparable?".

The next step will be to study the language-based relationships.

References

1. Abdulla, P.A., Nylén, A.: Timed petri nets and BQOs. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 53–70. Springer, Heidelberg (2001)
2. Aura, T., Lilius, J.: A causal semantics for time petri nets. Theoretical Computer Science 243(2), 409–447 (2000)
3. Berard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of the expressiveness of timed automata and time Petri nets. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, Springer, Heidelberg (2005)
4. Berthomieu, B.: La méthode des classes d'états pour l'analyse des réseaux temporels. mise en œuvre, extension à la multi-sensibilisation. In: Modélisation des Systémes Réactifs (MSR'01), Toulouse (Fr), pp. 275–290, (17–19 Octobre 2001)
5. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time petri nets. IEEE transactions on software engineering 17(3), 259–273 (March 1991)
6. Boyer, M., Vernadat, F.: Language and bisimulation relations between subclasses of timed petri nets with strong timing semantic. Technical report, LAAS (2000)
7. Boyer, M.: Translation from timed Petri nets with interval on transitions to interval on places (with urgency). In: Workshop on Theory and Practice of Timed Systems, vol. 65 of ENTCS, Grenoble, France, April 2002. Elsevier Science (2002)
8. Boyer, M., Diaz, M.: Multiple enabledness of transitions in time Petri nets. In: Proc. of the 9th IEEE International Workshop on Petri Nets and Performance Models, Aachen, Germany, September 11–14, 2001, IEEE Computer Society, Washington (2001)

9. Boyer, M., Roux, O. H.: Comparison of the expressiveness w.r.t. timed bisimilarity of k-bounded arc, place and transition time Petri nets with weak and strong single server semantics. Technical Report RI, -15, IRCCyN (2006) http://www.irccyn.ec-nantes.fr/hebergement/Publications/2006/3437.pdf
10. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: When are timed automata weakly timed bisimilar to time Petri nets? In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, Springer, Heidelberg (2005)
11. Cassez, F., Roux, O.H.: Structural translation from Time Petri Nets to Timed Automata – Model-Checking Time Petri Nets via Timed Automata. The journal of Systems and Software 79(10), 1456–1468 (2006)
12. Cerone, A., Maggiolo-Schettini, A.: Timed based expressivity of time petri nets for system specification. Theoretical Computer Science 216, 1–53 (1999)
13. de Frutos Escrig, D., Ruiz, V.V., Alonso, O.M.: Decidability of properties of timed-arc Petri nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 187–206. Springer, Heidelberg (2000)
14. Hanisch, H.M.: Analysis of place/transition nets with timed-arcs and its application to batch process control. In: Ajmone Marsan, M. (ed.) Application and Theory of Petri Nets 1993. LNCS, vol. 691, pp. 282–299. Springer, Heidelberg (1993)
15. Jones, N.D., Landweber, L.H., Lien, Y.E.: Complexity of some problems in Petri nets. Theoretical Computer Science 4, 277–299 (1977)
16. Khansa, W., Denat, J.-P., Collart-Dutilleul, S.: P-time Petri nets for manufacturing systems. In: International Workshop on Discrete Event Systems, WODES'96, Edinburgh (U.K.), pp. 94–102 (August 1996)
17. Khanza, W.: Réseau de Petri P-Temporels. Contribution á l'étude des systémes á événements discrets. PhD thesis, Université de Savoie (1992)
18. Merlin, P.M.: A study of the recoverability of computing systems. PhD thesis, Dep. of Information and Computer Science, University of California, Irvine, CA (1974)
19. Pezzè, M.: Time Petri Nets: A Primer Introduction. Tutorial presented at the Multi-Workshop on Formal Methods in Performance Evaluation and Applications, Zaragoza, Spain (September 1999)
20. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Project MAC Report MAC-TR-120 (1974)
21. Sifakis, J.: Performance Evaluation of Systems using Nets. In: Brauer, W. (ed.) Net Theory and Applications: Proc. of the advanced course on general net theory, processes and systems, LNCS, vol. 84, Springer, Heidelberg (1980)
22. Srba, J.: Timed-arc Petri nets vs. networks of timed automata. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 385–402. Springer, Heidelberg (2005)
23. Walter, B.: Timed net for modeling and analysing protocols with time. In: Proceedings of the IFIP Conference on Protocol Specification Testing and Verification, North Holland (1983)

Improving Static Variable Orders Via Invariants[*]

Gianfranco Ciardo[1], Gerald Lüttgen[2], and Andy Jinqing Yu[1]

[1] Dept. of Computer Science and Engineering, UC Riverside, CA 92521, USA
{ciardo,jqyu}@cs.ucr.edu

[2] Dept. of Computer Science, University of York, York YO10 5DD, U.K.
luettgen@cs.york.ac.uk

Abstract. Choosing a good variable order is crucial for making symbolic state-space generation algorithms truly efficient. One such algorithm is the MDD-based Saturation algorithm for Petri nets implemented in SMART, whose efficiency relies on exploiting event locality.

This paper presents a novel, static ordering heuristic that considers place invariants of Petri nets. In contrast to related work, we use the functional dependencies encoded by invariants to *merge* decision-diagram variables, rather than to eliminate them. We prove that merging variables always yields smaller MDDs and improves event locality, while eliminating variables may increase MDD sizes and break locality. Combining this idea of merging with heuristics for maximizing event locality, we obtain an algorithm for static variable order which outperforms competing approaches regarding both time-efficiency and memory-efficiency, as we demonstrate by extensive benchmarking.

1 Introduction

Petri nets [26] are a popular formalism for specifying concurrent and distributed systems, and much research [32] has been conducted in the automated analysis of a Petri net's state space. Many analysis techniques rely on generating and exploring a net's reachable markings, using algorithms based on *decision diagrams* [10,29] or *place invariants* [17,31,34,35].

While decision diagrams have allowed researchers to investigate real-world nets with thousands of places and transitions, their performance crucially depends on the underlying *variable order* [1,23]. Unfortunately, finding a good variable order is known to be an NP-complete problem [2]. Thus, many heuristics for either the static or the dynamic ordering of variables have been proposed, which have shown varying degree of success; see [18] for a survey.

In the state-space exploration of Petri nets, place invariants find use in approximating state spaces [28], since every reachable state must by definition satisfy each invariant, and in compactly storing markings by exploiting *functional dependencies* [6,19,27]. This latter use of invariants is also considered when encoding places with decision-diagram variables, as it eliminates some variables, offering hope for smaller decision diagrams during state-space exploration [17].

[*] Work supported in part by the National Science Foundation under grants CNS-0501747 and CNS-0501748 and by the EPSRC under grant GR/S86211/01.

The contributions of this paper are twofold. First, we show that *eliminating variables* based on invariance information may actually increase the sizes of decision diagrams, whence the above 'hope' is misplaced. Instead, we show that *merging variables* is guaranteed to lead to smaller decision diagrams. While our merging technique is obviously not applicable for *Binary* Decision Diagrams (BDDs), it is compatible with techniques using *Multi-way* Decision Diagrams (MDDs), such as SMART's *Saturation algorithm* for computing reachable markings [10]. In addition, merging variables improves *event locality*, i.e., it decreases the span of events over MDD levels, rather than worsening it as is the case with variable elimination. This is important since algorithms like Saturation become more efficient as event locality is increased.

Second, we propose a new *heuristic* for static variable ordering which is suitable for Saturation. This heuristic combines previous ideas, which only took the height and span of events into account [39], with variable merging based on linear place invariants. We implement our heuristic into SMART [9], generating the invariants with GreatSPN [7], and show via extensive benchmarking that this heuristic outperforms approaches that ignore place invariants, with respect to both time-efficiency and memory-efficiency. Indeed, the benefits of our heuristic are greatest for practical nets, including large instances of the *slotted-ring network* [30] and the *kanban system* [40], which have been tractable only using ad-hoc variable orderings and mergings found through our intuition and extensive experimentation. This shows that exploiting invariants is key for optimizing the performance of symbolic state-exploration techniques, provided one uses invariance information for merging variables and not for eliminating them.

Organization. The next section provides a short introduction to reachability and invariant analysis in Petri nets, and to decision diagrams and symbolic state-space generation. Sec. 3 recalls previous work on static variable ordering for Saturation, formally analyzes the concepts of variable elimination and merging, and develops our novel heuristic for static variable ordering. Sec. 4 experimentally evaluates our ideas on a suite of models. Finally, related work is discussed in Sec. 5, while Sec. 6 presents our conclusions and directions for future research.

2 Preliminaries

In this section we briefly cover the class of Petri nets considered, self-modifying nets, and their two main analysis approaches, reachability and invariant analysis. Then, we discuss decisions diagrams and how they can encode sets of markings and the transformations that transitions perform on markings. Finally, we survey a range of symbolic state-space generation algorithms, from the simple breadth-first iteration to our own Saturation algorithm.

2.1 Petri Nets and Self-modifying Nets

We consider *self-modifying nets with inhibitor arcs*, described by a tuple of the form $(\mathcal{P}, \mathcal{T}, \mathbf{F}^-, \mathbf{F}^+, \mathbf{F}^\circ, \mathbf{m}^{init})$, where

- $\mathcal{P}$ and $\mathcal{T}$ are sets of *places* and *transitions* satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$. A *marking* $\mathbf{m} \in \mathbb{N}^{\mathcal{P}}$ assigns a number of *tokens* $\mathbf{m}_p$ to each place $p \in \mathcal{P}$.
- $\mathbf{F}^{-}: \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \rightarrow \mathbb{N}$, $\mathbf{F}^{+}: \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \rightarrow \mathbb{N}$, and $\mathbf{F}^{\circ}: \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \rightarrow \mathbb{N} \cup \{\infty\}$ are $|\mathcal{P}| \times |\mathcal{T}|$ *incidence* matrices; $\mathbf{F}^{-}_{p,t}$, $\mathbf{F}^{+}_{p,t}$, and $\mathbf{F}^{\circ}_{p,t}$ are the *marking-dependent* [8,41] cardinalities of the *input*, *output*, and *inhibitor* arcs between p and t.
- $\mathbf{m}^{init}$ is the *initial* marking.

The evolution of the net from a marking $\mathbf{m}$ is governed by the following rules, keeping in mind that the cardinality of any arc is evaluated in the current marking, i.e., prior to the firing of any transition:

Enabling: Transition t is enabled in marking $\mathbf{m}$ if, for each place p, the input arc is satisfied, $\mathbf{m}_p \geq \mathbf{F}^{-}_{p,t}(\mathbf{m})$, and the inhibitor arc is not, $\mathbf{m}_p < \mathbf{F}^{\circ}_{p,t}(\mathbf{m})$.

Firing: Firing enabled transition t in marking $\mathbf{m}$ leads to marking $\mathbf{n}$, where, for each place p, $\mathbf{n}_p = \mathbf{m}_p - \mathbf{F}^{-}_{p,t}(\mathbf{m}) + \mathbf{F}^{+}_{p,t}(\mathbf{m})$. We write $\mathcal{N}_t(\mathbf{m}) = \{\mathbf{n}\}$, to stress that, for general discrete-state formalisms, the *next-state function* $\mathcal{N}_t$ for event t, applied to a single state $\mathbf{m}$, returns a set of states. Then, we can write $\mathcal{N}_t(\mathbf{m}) = \emptyset$ to indicate that t is not enabled in marking $\mathbf{m}$.

2.2 Reachability Analysis and Invariant Analysis

The two main techniques for Petri net analysis are *reachability analysis* and *invariant analysis*. The former builds and analyzes the *state space* of the net (or *reachability set*), defined as $\mathcal{M} = \{\mathbf{m} : \exists d, \mathbf{m} \in \mathcal{N}^d(\mathbf{m}^{init})\} = \mathcal{N}^*(\mathbf{m}^{init})$, where we extend the next-state function to arbitrary sets of markings $\mathcal{X} \subseteq \mathbb{N}^{\mathcal{P}}$, $\mathcal{N}_t(\mathcal{X}) = \bigcup_{\mathbf{m} \in \mathcal{X}} \mathcal{N}_t(\mathbf{m})$, write $\mathcal{N}$ for the union of all next-state functions, $\mathcal{N}(\mathcal{X}) = \bigcup_{t \in \mathcal{T}} \mathcal{N}_t(\mathcal{X})$, and define multiple applications of the next-state function as usual, $\mathcal{N}^0(\mathcal{X}) = \mathcal{X}$, $\mathcal{N}^d(\mathcal{X}) = \mathcal{N}(\mathcal{N}^{d-1}(\mathcal{X}))$, and $\mathcal{N}^*(\mathcal{X}) = \bigcup_{d \in \mathbb{N}} \mathcal{N}^d(\mathcal{X})$.

Invariant analysis is instead concerned with deriving *a priori* relationships guaranteed to be satisfied by any reachable marking, based exclusively on the net structure. In nets where the arcs have a constant cardinality independent of the marking, i.e., ordinary Petri nets with or without inhibitor arcs [26], much work has focused on the computation of *p-semiflows* [14,15], i.e., non-zero solutions $\mathbf{w} \in \mathbb{N}^{\mathcal{P}}$ to the linear set of "flow" equations $\mathbf{w} \cdot \mathbf{F} = \mathbf{0}$, where $\mathbf{F} = \mathbf{F}^{+} - \mathbf{F}^{-}$. Since any linear combination of such solutions is also a solution, it suffices to consider a set of *minimal* p-semiflows from which all others can be derived through non-negative linear combinations. A semiflow $\mathbf{w}$ specifies the constraint $\sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \mathbf{m}_p = C$ on any reachable marking $\mathbf{m}$, where the constant $C = \sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \mathbf{m}^{init}_p$ is determined by the initial marking. When marking-dependent arc multiplicities are present, linear p-semiflows [8], or even more general relationships [41], may still exist. However, invariant analysis provides necessary, not sufficient, conditions on reachability; a marking $\mathbf{m}$ might satisfy all known invariants and still be unreachable.

In this paper, we use invariants to improve (symbolic) state-space generation. We assume to be given a self-modifying net with inhibitor arcs (or a similar discrete-state model whose next-state function is decomposed according to a set of asynchronous events), and a set $\mathcal{W}$ of linear invariants, each of the form $\sum_{p\in\mathcal{P}} \mathbf{W}_{v,p} \cdot \mathbf{m}_p = C_v$, guaranteed to hold in any reachable marking $\mathbf{m}$. Then,

– $\mathit{Support}(v) = \{p \in \mathcal{P} : \mathbf{W}_{v,p} > 0\}$ is the *support* of the v^{th} invariant.
– $\mathbf{W} \in \mathbb{N}^{|\mathcal{W}|\times|\mathcal{P}|}$ describes the set of invariants. In addition, observe that the case $|\mathit{Support}(v)| = 1$ is degenerate, as it implies that the marking of the place p in the support is fixed. We then assume that p is removed from the net after modifying it appropriately, i.e., substituting the constant $\mathbf{m}_p^{init}$ for $\mathbf{m}_p$ in the marking-dependent expression of any arc and removing any transition t with $\mathbf{F}^-_{p,t} > \mathbf{m}_p^{init}$ or $\mathbf{F}^\circ_{p,t} \le \mathbf{m}_p^{init}$. Thus, each row of $\mathbf{W}$ contains at least two positive entries.
– The marking of any one place $p \in \mathit{Support}(v)$ can be expressed as a function of the places in $\mathit{Support}(v) \setminus p$ through inversion, i.e., in every reachable marking $\mathbf{m}$, the relation $\mathbf{m}_p = (C_v - \sum_{q\in\mathcal{P}\setminus\{p\}} \mathbf{W}_{v,q} \cdot \mathbf{m}_q)/\mathbf{W}_{v,p}$ holds.

We say that a set of non-negative integer variables $\mathcal{V}'$ is *functionally dependent* on a set of non-negative integer variables $\mathcal{V}''$ if, when the value of the variables in $\mathcal{V}''$ is known, the value of the variables in $\mathcal{V}'$ is uniquely determined. In our linear Petri-net invariant setting, $\mathcal{V}'$ and $\mathcal{V}''$ correspond to the markings of two sets of places $\mathcal{P}'$ and $\mathcal{P}''$, and functional dependence implies that the submatrix $\mathbf{W}_{\mathcal{W}',\mathcal{P}'}$ of $\mathbf{W}$, obtained by retaining only columns corresponding to places in $\mathcal{P}'$ and rows corresponding to invariants having support in $\mathcal{P}' \cup \mathcal{P}''$, i.e., $\mathcal{W}' = \{v \in \mathcal{W} : \mathit{Support}(v) \subseteq \mathcal{P}' \cup \mathcal{P}''\}$, has rank $|\mathcal{P}'|$. This fundamental concept of functional dependence is at the heart of our treatment, and could be generalized to the case of nonlinear invariants where not every place in $\mathit{Support}(v)$ can be expressed as a function of the remaining places in the support. To keep presentation and notation simple, we do not discuss such invariants.

2.3 Decision Diagrams

The state-space generation algorithms we consider use *quasi-reduced ordered multi-way decision diagrams* (MDDs) [22] to store *structured* sets, i.e., subsets of a *potential set* $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$, where each *local set* $\mathcal{S}_l$, for $K \ge l \ge 1$, is of the form $\{0, 1, ..., n_l - 1\}$. Formally, an MDD over $\widehat{\mathcal{S}}$ is a directed acyclic edge-labeled multi-graph such that:

– Each node p belongs to a *level* in $\{K, ..., 1, 0\}$, denoted $p.lvl$.
– There is a single *root* node $r^\star$ at level K or 0.
– Level 0 can contain only the *terminal* nodes $\mathbf{0}$ and $\mathbf{1}$.
– A node p at level $l > 0$ has n_l outgoing edges, labeled from 0 to $n_l - 1$. The edge labeled by $i \in \mathcal{S}_l$ points to node q at level $l - 1$ or 0; we write $p[i] = q$.
– Given nodes p and q at level l, if $p[i] = q[i]$ for all $i \in \mathcal{S}_l$, then $p = q$.
– The edges of a node at level $l > 0$ cannot all point to $\mathbf{0}$ or all point to $\mathbf{1}$.

An MDD node p at level l encodes, with respect to level $m \ge l$, the set of tuples $\mathcal{B}(m, p) = \mathcal{S}_m \times \cdots \times \mathcal{S}_{l+1} \times \left(\bigcup_{i\in\mathcal{S}_l} \{i\} \times \mathcal{B}(l - 1, p[i])\right)$, letting $\mathcal{X} \times \mathcal{B}(0, \mathbf{0}) = \emptyset$ and $\mathcal{X} \times \mathcal{B}(0, \mathbf{1}) = \mathcal{X}$. If $m = l$, we write $\mathcal{B}(p)$ instead of $\mathcal{B}(l, p)$. Fig. 1 contains

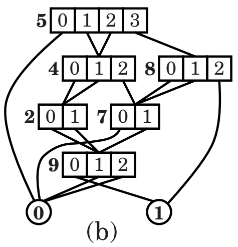
$\mathcal{S}_4 = \{0, 1, 2, 3\}$

$\mathcal{S}_3 = \{0, 1, 2\}$

$\mathcal{S}_2 = \{0, 1\}$

$\mathcal{S}_1 = \{0, 1, 2\}$

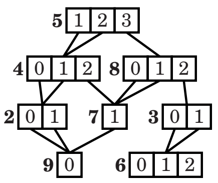
(a)



(b)

$\mathcal{S} = \{1000, 1010, 1100, 1110, 1210, 2000, 2010, 2100, 2110, 2210, 3010, 3110, 3200, 3201, 3202, 3210, 3211, 3212\}$

(c)



(d)

Fig. 1. An example of an MDD and the set of 4-tuples it encodes

an example where $K = 4$, showing the composition of the sets $\mathcal{S}_l$ (a), the MDD (b), and the set of tuples encoded by it (c). Here, as in [11], we employ a *dynamic* MDD variant where the sets $\mathcal{S}_l$ are not fixed but are grown as needed, so that the MDD can be used to encode arbitrary (but finite) subsets of $\mathbb{N}^K$. The only overhead in such a data structure is that, since a set $\mathcal{S}_l$ may grow and change the meaning of an edge spanning level l and pointing to node **1**, only edges pointing to node **0** are allowed to span multiple levels, while node **1** can be pointed only by edges from nodes at level 1. Fig. 1(d) shows how this requires the insertions of nodes **3** and **6** along edge 2 from **8**; we employ a simplified representation style where terminal nodes and edges to **0** are omitted.

For our class of nets, it might be difficult (and it is generally impossible) to compute an upper bound on the marking of a place. To store a set of reachable markings during symbolic state-space generation, we could then use dynamic MDDs over $\mathbb{N}^{|\mathcal{P}|}$, so that a marking $\mathbf{m}$ is simply a tuple encoded in the MDD. However, this simplistic approach has several drawbacks:

– Even if the (current) bound B_p on the marking of a place p is tight, i.e., there is a reachable marking $\mathbf{m}$ with $\mathbf{m}_p = B_p$, the local set $\mathcal{S}_p$ might have "holes", i.e., no reachable marking $\mathbf{n}$ might have $\mathbf{n}_p = c$, for some $0 \leq c < B_p$. This may waste memory or computation during MDD manipulation.
– If many different markings for p are possible, $\mathcal{S}_p$ and thus the nodes at level p, might be too large, again decreasing the efficiency of MDD manipulations. It might then be better to *split* a place over multiple MDD levels. This is actually necessary if the implementation uses BDDs [3], which are essentially our MDDs restricted to the case where each $\mathcal{S}_l$ is just $\{0, 1\}$.
– On the other hand, our symbolic algorithms can greatly benefit from "event locality" which we discuss later. To enhance such locality, we might instead want to *merge* certain places into a single MDD level.
– If some invariants are known, we can avoid storing some of the $|\mathcal{P}|$ components of the marking, since they can be recovered from the remaining ones.

For simplicity, and since we employ MDDs, we ignore the issue of splitting a place over multiple levels, but assume the use of $K \leq |\mathcal{P}|$ *indexing* functions that map submarkings into natural numbers. Given a net, we partition its places into K subsets $\mathcal{P}_K, ..., \mathcal{P}_1$, so that a marking $\mathbf{m}$ is written as the collection of the K submarkings $(\mathbf{m}_K, ..., \mathbf{m}_1)$. Then, $\mathbf{m}$ can be mapped to the tuple of the corresponding K submarking indices $(\mathbf{i}_K, ..., \mathbf{i}_1)$, where $\mathbf{i}_l = \psi_l(\mathbf{m}_l)$ and $\psi_l : \mathbb{N}^{|\mathcal{P}_l|} \to \mathbb{N} \cup \{\mathsf{null}\}$ is a partial function. In practice, each ψ_l only needs to map the set $\mathcal{M}_l$ of submarkings for $\mathcal{P}_l$ known to be reachable so far to the range $\{0, ..., |\mathcal{M}_l| - 1\}$ of natural numbers. We can then define ψ_l *dynamically*:

– Initially, set $\mathcal{M}_l = \{\mathbf{m}_l^{init}\}$ and $\psi_l(\mathbf{m}_l^{init}) = 0$, i.e., map the only known submarking for $\mathcal{P}_l$, the initial submarking, to the first natural number.
– For any other $\mathbf{m}_l \in \mathbb{N}^{|\mathcal{P}_l|} \setminus \mathcal{M}_l$, i.e., any submarking not yet discovered, set $\psi_l(\mathbf{m}_l)$ to the default value null.
– When a new submarking $\mathbf{m}_l$ for $\mathcal{P}_l$ is discovered, add it to $\mathcal{M}_l$ and set $\psi_l(\mathbf{m}_l) = |\mathcal{M}_l| - 1$.

This mapping can be efficiently stored in a search tree and offers great flexibility in choosing the MDD variables $(\mathbf{x}_K, ..., \mathbf{x}_1)$ corresponding to the possible values of the indices at each level. We can have as little as a single variable (when $K = 1$, $\mathcal{S}_1 = \mathcal{S}$ and we perform an explicit reachability set generation), or as many as $|\mathcal{P}|$ variables, so that each place corresponds to a different level of the MDD.

2.4 Symbolic Algorithms to Generate the State Space of a Net

We now focus on building the state space of a net using MDDs, i.e., on computing $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ corresponding to $\mathcal{M}$. Since the functions ψ_l, $K \geq l \geq 1$, provide a bijection between markings and K-tuples, knowledge of $\mathcal{S}$ implies knowledge of $\mathcal{M}$. As they manipulate sets of tuples, not individual tuples, all symbolic state-space generation algorithms are some variation of the following:

> "Build the MDD encoding $\mathcal{S}$, defined as the smallest solution to the fixpoint equation $\mathcal{S} = \mathcal{S} \cup \mathcal{N}(\mathcal{S})$ subject to $\mathcal{S} \supseteq \mathcal{S}^{init}$",

where the next-state function $\mathcal{N}$ is now applied to tuples instead of markings.

Of course, $\mathcal{N}$ is also encoded using either MDDs or related formalisms. The most common choice is a $2K$-level MDD with *interleaved* levels for the *current* variables $\mathbf{x}$ and the *next* variables $\mathbf{x}'$, i.e., if $\mathbf{i}' \in \mathcal{N}(\mathbf{i})$, there is a path $(i_K, i'_K, ..., i_1, i'_1)$ from the root of the MDD encoding $\mathcal{N}$ to node $\mathbf{1}$. In our asynchronous context, a *disjunctive partition* [4] can be used, where each transition $t \in \mathcal{T}$ is encoded as a separate $2K$-level MDD. This is the case in the standard breadth-first algorithm *Bfs* shown in Fig. 2. Function *Union* returns the root of the MDD encoding the union of the sets encoded by the arguments (all encoded as K-level MDDs), while function *Image* returns the root of the MDD encoding the set of states reachable in one application of the second argument (a $2K$-level MDD) from any state encoded by the first argument (a K-level MDD); both functions are easily expressed in recursive form. In the figure, we identify sets and relations with the MDDs encoding them; thus, for example, $\mathcal{N}_t[i][i']$ means the node in the MDD encoding $\mathcal{N}_t$ which is reached by following the edge labeled i from the root and then the edge labeled i' from the resulting node.

To improve over breadth-first iterations, we have proposed algorithms [12,13] that exploit *chaining* [33] and *event locality*. Chaining is based on the observation

```
mdd $Bfs$( mdd $\mathcal{S}^{init}$ )
1 $\mathcal{S} \leftarrow \mathcal{S}^{init}$;
2 repeat
3    $\mathcal{X} \leftarrow \emptyset$;
4    foreach $t \in \mathcal{T}$ do
5       $\mathcal{X} \leftarrow Union(\mathcal{X}, Image(\mathcal{S}, \mathcal{N}_t))$;
6    $\mathcal{S} \leftarrow Union(\mathcal{S}, \mathcal{X})$;
7 until $\mathcal{S}$ does not change;
8 return $\mathcal{S}$;
```

```
mdd $BfsChaining$( mdd $\mathcal{S}^{init}$ )
1 $\mathcal{S} \leftarrow \mathcal{S}^{init}$;
2 repeat
3    foreach $t \in \mathcal{T}$ do
4       $\mathcal{S} \leftarrow Union(\mathcal{S}, Image(\mathcal{S}, \mathcal{N}_t))$;
5 until $\mathcal{S}$ does not change;
6 return $\mathcal{S}$;
```

```
void $Saturation$( mdd $\mathcal{S}^{init}$ )
1 for $l = 1$ to $K$ do
2    foreach node $p$ at level $l$ on a path from $\mathcal{S}^{init}$ to $\mathbf{1}$ do
3       $Saturate(p)$;                                   • update $p$ in place
```

```
void $Saturate$( mdd $p$ )
1 $l \leftarrow p.lvl$;
2 repeat
3    choose $t$ s.t. $Top(t) = l$, $i \in \mathcal{S}_l, i' \in \mathcal{S}_l$ s.t. $p[i] \neq \mathbf{0}$ and $\mathcal{N}_t[i][i'] \neq \mathbf{0}$;
4    $p[i'] \leftarrow Union(p[i'], ImageSat(p[i], \mathcal{N}_t[i][i']))$;
5 until $p$ does not change;
```

```
mdd $ImageSat$( mdd $q$, mdd2 $f$ )
1 if $q = \mathbf{0}$ or $f = \mathbf{0}$ then return $\mathbf{0}$;
2 $k \leftarrow q.lvl$;                                   • $f.lvl = k$ as well
3 $s \leftarrow$ new level-$k$ node with edges set to $\mathbf{0}$;
4 foreach $i \in \mathcal{S}_k, i' \in \mathcal{S}_k$ s.t. $q[i] \neq \mathbf{0}$ and $f[i][i'] \neq \mathbf{0}$ do
5    $s[i'] \leftarrow Union(s[i'], ImageSat(q[i], f[i][i']))$;
6 $Saturate(s)$;
7 return $s$.
```

Fig. 2. Breadth-first, chaining, and Saturation state-space generation

that the number of symbolic iterations might be reduced if the application of asynchronous events (transitions) is compounded sequentially, see *BfsChaining* in Fig. 2. While the search order is not strictly breadth-first anymore, the set of known states at the d^{th} iteration of the repeat-until loop is guaranteed to be at least as large with chaining as without.

However, the efficiency of symbolic state-space generation is determined not just by the *number* of iterations but also by their *cost*, i.e., by the size of the MDDs involved. In practice, chaining has been shown to be quite effective in many asynchronous models, but its effectiveness can be greatly affected by the order in which transitions are applied. Event locality can then be used to define a good ordering heuristic [12], as we explain next.

Given a transition t, we define $\mathcal{V}_M(t) = \{x_l : \exists \mathbf{i}, \mathbf{i}' \in \widehat{\mathcal{S}}, \mathbf{i}' \in \mathcal{N}_t(\mathbf{i}) \wedge i_l \neq i'_l\}$ and $\mathcal{V}_D(t) = \{x_l : \exists \mathbf{i}, \mathbf{j} \in \widehat{\mathcal{S}}, \forall k \neq l, i_k = j_k \wedge \mathcal{N}_t(\mathbf{i}) \neq \emptyset \wedge \mathcal{N}_t(\mathbf{j}) = \emptyset\}$, i.e., the variables that can be modified by t, or that can disable, t, respectively. Moreover, we let $Top(t) = \max\{l : x_l \in \mathcal{V}_M(t) \cup \mathcal{V}_D(t)\}$ and $Bot(t) = \min\{l : x_l \in \mathcal{V}_M(t) \cup \mathcal{V}_D(t)\}$.

We showed experimentally in [12] that applying the transitions $t \in \mathcal{T}$ in an order consistent with their value of Top, from 1 to K, results in effective chaining.

Locality is easily determined for our nets since the enabling and firing effect of a transition t depend only on its input, output, and inhibitor places, plus any place appearing in the cardinality expression of the corresponding arcs.

Recognizing locality, however, offers great potential beyond suggesting a good chaining order. If $Top(t) = l$ and $Bot(t) = k$, any variable x_m outside this range, i.e., above l or below k, is not changed by the firing of transition t. When computing the image in line 4 of *BfsChaining*, we can then access only MDD nodes at level l or below and update *in-place* only MDD nodes at level l, without having to access the MDD from the root. While *Kronecker* [25] and *matrix diagram* [24] encodings have been used to exploit these *identity transformations* in $\mathcal{N}_t$, the most general and efficient data structure appears to be a decision diagram with special reduction rules [13]. In this paper, we assume that $\mathcal{N}_t$ is encoded with an MDD over just the current and next variables between $Top(t)$ and $Bot(t)$ included, instead of a $2K$-level MDD. If there are no marking-dependent arc cardinalities, the structure of this MDD is quite simple, as we simply need to encode the effect of every input, inhibitor, and output arc connected to t; the result is an MDD with just one node per 'unprimed' level. For general self-modifying nets, a "localized" explicit enumeration approach may be used [13], although a completely symbolic approach might be preferable.

We can now introduce our most advanced algorithm, *Saturation*, also shown in Fig. 2. An MDD node p at level l is *saturated* [10] if

$$\forall t \in \mathcal{T},\ \ Top(t) \leq l \ \Rightarrow\ \mathcal{B}(K,p) \supseteq \mathcal{N}_t(\mathcal{B}(K,p)).$$

To saturate node p once its descendants are saturated, we compute the effect of firing t on p for each transition t such that $Top(t) = l$, recursively saturating any node at lower levels created in the process, and add the result to $\mathcal{B}(p)$ using in-place updates. Thus *Saturation* proceeds saturating the nodes in the MDD encoding the initial set of states bottom-up, starting at level 1 and stopping when the root at level K is saturated.

Only saturated nodes appear in the *operation cache* (needed to retrieve the result of an *ImageSat* or *Union* call, if it has already been issued before with the same parameters) and the *unique table* (needed to enforce MDD canonicity by recognizing duplicate nodes). Since nodes in the MDD encoding the final $\mathcal{S}$ are saturated by definition, this unique property, not shared by any other approach, is key to a much greater efficiency. Indeed, we have experimentally found that both memory and run-time requirements for our Saturation approach are usually several orders of magnitude smaller than for the traditional symbolic breadth-first exploration, when modeling asynchronous systems.

3 Structural Invariants to Improve Symbolic Algorithms

Structural invariants have been proposed for *variable elimination.* For example, [17] suggests an algorithm that starts with an empty set of boolean variables (places of a safe Petri net) and examines each place in some arbitrary order, adding it as new (lower) level of the BDD only if it is not functionally dependent on the current set of variables. This greedy elimination algorithm reduces the number of levels of the BDD, with the goal of making symbolic state-space generation more efficient. However, we argue that this invariant-based elimination severely hurts locality and is generally a bad idea, not only for Saturation, but even for the simpler BFS iterations (if properly implemented to exploit locality). To see why this is the case, consider a transition t with an input or inhibitor arc from a place p, i.e., $p \in \mathcal{V}_D(t)$. If p is in the support of invariant v and is eliminated because all other places in $Support(v)$ already correspond to BDD levels, the marking of p can indeed be determined from the marking of each place $q \in Support(v) \setminus \{p\}$. However, this not only removes p from $\mathcal{V}_D(t)$ but also adds $Support(v) \setminus \{p\}$ to it. In most cases, the *span* of transition t, i.e., the value of $Top(t) - Bot(t) + 1$, can greatly increase, resulting in a more costly image computation for $\mathcal{N}_t$.

The solution we present in Sec. 3.1, enabled by our use of MDDs instead of BDDs, is to perform instead *variable merging*. This achieves the same goal of reducing the number of levels (actually resulting in more levels being eliminated, since it considers groups of variables at a time, not just individual ones as in [17]), without negatively affecting locality and actually improving it for a meaningful class of nets. Then, having at our disposal the invariants, we turn to the problem of variable ordering, and show in Sec. 3.2 how our previous idea of minimizing the sum of the top levels affected by each transition [39] can be extended to take into account invariants as well, treating an invariant analogously to a transition and its support as if it were the set of places "affected" by the invariant.

3.1 Using Structural Invariants to Merge State Variables

As one of our main contributions, we first present and prove a general theorem stating that *merging two MDD variables* based on functional dependence guarantees to reduce the size of an MDD. In contrast, we show that placing variables in the support of an invariant close to each other without merging them, as suggested by many static and dynamic variable reordering techniques [21,19], may actually increase the size of the MDD. We then adopt our merging theorem to improve Petri-net state-space encodings with place invariants, and present a greedy algorithm to iteratively merge MDD variables given a set of place invariants obtained from a structural analysis of a net. Thus, our goal is to determine both a merging of the MDD variables and an ordering of these merged variables.

Variable merging based on functional dependence. To discuss what happens to the size of an MDD when merging variables based on functional dependence, one must take into account both the number of nodes and their sizes. To be precise, and to follow what is done in an efficient "sparse node" implementation, the size of an MDD node is given by the number of its non-zero edges, i.e., the number of outgoing edges that do not point to node $\mathbf{0}$. Thus, since a node has always at least one non-zero edge, the size of a node for variable x_l can range from one to $|\mathcal{S}_l|$. First, we recall a theorem on the number of MDD nodes required to encode a boolean function f.

Theorem 1. [38] Using the variable order $(x_K, ..., x_1)$, the number of MDD nodes for variable $x_l \in \mathcal{S}_l$, for $K \geq l \geq 1$, in the MDD encoding of a boolean function $f(x_K, ..., x_1)$ equals the number of different subfunctions obtained by fixing the values of $x_K, ..., x_{l+1}$ to all the possible values $i_K \in \mathcal{S}_K, ..., i_{l+1} \in \mathcal{S}_{l+1}$.

In the following, we denote by $f[^{x_{k_1}}_{i_{k_1}}, ..., ^{x_{k_n}}_{i_{k_n}}]$ the subfunction obtained from f by fixing the value of $x_{k_1}, ..., x_{k_n}$ to $i_{k_1}, ..., i_{k_n}$, and we use the same symbol for a boolean function and its MDD encoding, once the variable order is given.

Theorem 2. Consider an MDD f encoding a set $\mathcal{X} \subseteq \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ with variable order $\pi = (x_K, ..., x_1)$. If x_m is functionally dependent on $\{x_K, ..., x_k\}$, with $k > m$, then define a new variable $x_{k,m} \equiv x_k n_m + x_m$, where $n_m = |\mathcal{S}_m|$, having domain $\mathcal{S}_{k,m} = \{0, ..., |\mathcal{S}_k \times \mathcal{S}_m| - 1\}$. Let the MDD g encode $\Lambda(\mathcal{X})$ with variable order $\overline{\pi} = (x_K, ..., x_{k+1}, x_{k,m}, x_{k-1}, ..., x_{m+1}, x_{m-1}, ..., x_1)$, where $\Lambda(x_K, ..., x_1) = (x_K, ..., x_{k+1}, x_{k,m}, x_{k-1}, ..., x_{m+1}, x_{m-1}, ..., x_1)$.

Then, (1) $f[^{x_k}_{i_k}, ^{x_m}_{i_m}] \equiv g[^{x_{k,m}}_{i_k n_m + i_m}]$; and (2) g requires strictly fewer MDD nodes and non-zero edges than f.

Proof. Property (1) follows directly from the definition of $x_{k,m}$, Λ, and g. Let $\overline{\nu_l}$ and ν_l be the number of nodes corresponding to variable x_l in g and f, respectively. Analogously, let $\overline{\epsilon_l}$ and ϵ_l be the number of non-zero edges leaving these nodes. To establish Property (2), we prove that $\overline{\nu_{k,m}} = \nu_k$ and $\overline{\epsilon_{k,m}} = \epsilon_k$, $\overline{\nu_l} = \nu_l$ and $\overline{\epsilon_l} = \epsilon_l$ for $x_l \in \{x_K, ..., x_{k+1}, x_{m-1}, ..., x_1\}$, and $\overline{\nu_l} \leq \nu_l$ and $\overline{\epsilon_l} \leq \epsilon_l$ for $x_l \in \{x_{k-1}, ..., x_{m+1}\}$. These relations, in addition to the fact that f contains additional $\nu_m > 0$ nodes corresponding to x_m (each of them having exactly one non-zero edge, because of the functional dependence), show that g is encoded with at least $\nu_m = \epsilon_m$ fewer nodes and edges than f. We now prove these relations by considering the different possible positions of variable x_l in $\overline{\pi}$.

Case 1: $x_l \in \{x_{m-1}, ..., x_1\}$. Since $f[^{x_k}_{i_k}, ^{x_m}_{i_m}] \equiv g[^{x_{k,m}}_{i_k n_m + i_m}]$, we let f_1 and g_1 be

$$f_1 = f[^{x_k}_{i_k}, ^{x_m}_{i_m}][^{x_K}_{i_K}, ..., ^{x_{k+1}}_{i_{k+1}}, ^{x_{k-1}}_{i_{k-1}}, ..., ^{x_{m+1}}_{i_{m+1}}, ^{x_{m-1}}_{i_{m-1}}, ..., ^{x_{l+1}}_{i_{l+1}}]$$
$$g_1 = g[^{x_{k,m}}_{i_k n_m + i_m}][^{x_K}_{i_K}, ..., ^{x_{k+1}}_{i_{k+1}}, ^{x_{k-1}}_{i_{k-1}}, ..., ^{x_{m+1}}_{i_{m+1}}, ^{x_{m-1}}_{i_{m-1}}, ..., ^{x_{l+1}}_{i_{l+1}}]$$

and conclude that $f_1 \equiv g_1$. Recall that the number of nodes of variable x_l in f is the number of different subfunctions $f[^{x_K}_{i_K}, ..., ^{x_{l+1}}_{i_{l+1}}]$, for all possible $i_K, ..., i_{l+1}$. Since f and g have the same set of such subfunctions, we must have $\overline{\nu_l} = \nu_l$. To see that $\overline{\epsilon_l} = \epsilon_l$ as well, simply observe that each pair of corresponding MDD nodes, e.g., f_1 and g_1, must have the same number of non-zero edges, since $f_1 \equiv g_1$ implies $f_1[^{x_l}_{i_l}] \equiv g_1[^{x_l}_{i_i}]$ for any $i_l \in \mathcal{X}_l$, and the edge i_l is non-zero if and only if $f_1[^{x_l}_{i_l}] \not\equiv \mathbf{0}$.

Case 2: $x_l \in \{x_{k-1}, ..., x_{m+1}\}$. Consider two different nodes of x_l in g, encoding two *different* subfunctions g_1 and g_2, which obviously satisfy $g_1 \not\equiv 0$ and $g_2 \not\equiv 0$:

$$g_1 \equiv g[^{x_K}_{i_K}, ..., ^{x_{k+1}}_{i_{k+1}}, ^{x_{k,m}}_{i_k n_m + i_m}, ^{x_{k-1}}_{i_{k-1}}, ..., ^{x_{l+1}}_{i_{l+1}}] \quad g_2 \equiv g[^{x_K}_{j_K}, ..., ^{x_{k+1}}_{j_{k+1}}, ^{x_{k,m}}_{j_k n_m + j_m}, ^{x_{k-1}}_{j_{k-1}}, ..., ^{x_{l+1}}_{j_{l+1}}].$$

Then, define f_1 and f_2 as follows, obviously satisfying $f_1 \not\equiv 0$ and $f_2 \not\equiv 0$, too:

$$f_1 \equiv f[^{x_K}_{i_K}, ..., ^{x_{k+1}}_{i_{k+1}}, ^{x_k}_{i_k}, ^{x_{k-1}}_{i_{k-1}}, ..., ^{x_{l+1}}_{i_{l+1}}] \qquad f_2 \equiv f[^{x_K}_{j_K}, ..., ^{x_{k+1}}_{j_{k+1}}, ^{x_k}_{j_k}, ^{x_{k-1}}_{j_{k-1}}, ..., ^{x_{l+1}}_{i_{l+1}}].$$

We prove by contradiction that f_1 and f_2 must be different and therefore encoded by two different nodes of variable x_l in f. Since x_m is functionally dependent on $\{x_K, ..., x_k\}$ and the value of $(x_K, ..., x_k)$ is fixed to $(i_K, ..., i_k)$ for f_1 and to $(j_K, .., j_1)$ for f_2, there must exist unique values i_m and j_m such that $f_1[^{x_m}_{i_m}] \not\equiv 0$ and $f_2[^{x_m}_{j_m}] \not\equiv 0$. If f_1 and f_2 were the same function, we would have $i_m = j_m$ and $f_1[^{x_m}_{i_m}] \equiv f_2[^{x_m}_{j_m}]$. From Property (1), we then obtain $g_1 \equiv f_1[^{x_m}_{i_m}] \equiv f_2[^{x_m}_{j_m}] \equiv g_2$, a contradiction. Thus, distinct nodes of g must correspond to distinct nodes of f, i.e., $\overline{\nu_l} \leq \nu_l$. Again, to see that $\overline{\epsilon_l} \leq \epsilon_l$, observe that the MDD nodes encoding f_1 and g_1 must have the same number of non-zero edges because, for all $i_l \in \mathcal{S}_l$, $g_1[^{x_l}_{i_l}] \equiv f_1[^{x_m}_{i_m}][^{x_l}_{i_l}]$. Furthermore, if multiple nodes in f correspond to the same node of g, i.e., if $\overline{\nu_l} < \nu_l$, we also have $\overline{\epsilon_l} < \epsilon_l$.

Case 3: $x_l \in \{x_K, ..., x_{k+1}, x_{k,m}\}$. Observe that $g \equiv \Lambda(f)$ and $g[^{x_K}_{i_K}, ..., ^{x_{l+1}}_{i_{l+1}}] \equiv \Lambda_l(f[^{x_K}_{i_K}, ..., ^{x_{l+1}}_{i_{l+1}}])$, where Λ_l is defined analogously to Λ, i.e., $\Lambda_l(x_l, ..., x_1) = (x_l, x_{k+1}, x_{k+m}, x_{k-1}, ..., x_{m+1}, ..., x_{m-1}, ..., x_1)$, As for Case 1, we can prove that $\overline{\nu_l} = \nu_l$ and $\overline{\epsilon_l} = \epsilon_l$ by observing that g and f must have the same subfunctions and the MDD nodes encoding these subfunctions must have the same number of non-zero edges. □

Intuitively, merging variable x_m with x_k is not that different from moving it just below x_k in the variable order, the commonly suggested approach for BDDs to help reduce the number of nodes [19,21]. However, the example in Fig. 3 illustrates that the latter can instead increase the BDD size. Fig. 3(a) shows an example of MDDs that encodes a boolean function with initial variable order (d, c, b, a), satisfying the invariant $a + c + d = 2$. Fig. 3(b) shows the result of reordering the MDD to put variables a, c, and d close to each other, by swapping variables b and a. Note that the number of nodes in the second MDD increases from six to seven, and the number of non-zero edges from seven to eight. Fig. 3(c) shows instead the result of merging variables a and c, where the number of nodes decreases from six to five and the number of non-zero edges from seven to six, as predicted by Thm. 2. The meaning of the elements of $\mathcal{S}_l$ in terms of the value of the variables assigned to level l is shown to the right of each MDD. We stress that this reduction in the number of nodes can only be achieved if the MDDs are implemented natively, not as the interface to BDDs implemented in [22,36]; this is apparent since Fig. 3(b) is exactly the BDD that would be built if the MDD of Fig. 3(c) were implemented using BDDs.

Focusing now on Petri nets, we restate Thm. 2 in their terminology and use place invariants to determine functional dependence.

Theorem 3. Consider a Petri net with an ordered partition π of $\mathcal{P}$ into the sets $(\mathcal{P}_K, ..., \mathcal{P}_1)$ and mappings $\psi_{\mathcal{P}_l} : \mathbb{N}^{\mathcal{P}_l} \to \mathbb{N} \cup \{\mathsf{null}\}$, for $K \geq l \geq 1$. Let the ordered partition $\overline{\pi}$ be the one obtained by merging $\mathcal{P}_k$ and $\mathcal{P}_m$ into $\mathcal{P}_{\{k,m\}}$, with $k > m$, resulting in the order $(\mathcal{P}_K, ..., \mathcal{P}_{\{k,m\}}, \mathcal{P}_{k-1}, ..., \mathcal{P}_{m+1}, \mathcal{P}_{m-1}, ..., \mathcal{P}_1)$ and the same mappings as before, except for the new $\psi_{\{\mathcal{P}_k,\mathcal{P}_m\}} : \mathbb{N}^{\mathcal{P}_{\{k,m\}}} \to \mathbb{N} \cup \{\mathsf{null}\}$ to replace ψ_k and ψ_m, which must satisfy $\psi_{\{\mathcal{P}_k,\mathcal{P}_m\}}(\mathbf{m}_{\mathcal{P}_k}, \mathbf{m}_{\mathcal{P}_m}) = \mathsf{null}$ if and only if $\psi_k(\mathbf{m}_{\mathcal{P}_k}) = \mathsf{null}$ or $\psi_m(\mathbf{m}_{\mathcal{P}_m}) = \mathsf{null}$. Then, if $\mathcal{P}_m$ is functionally dependent on $\bigcup_{K \geq l \geq k} \mathcal{P}_l$, the MDD encoding of any nonempty set of markings $\mathcal{X}$ requires strictly fewer nodes and edges with $\overline{\pi}$ than with π.

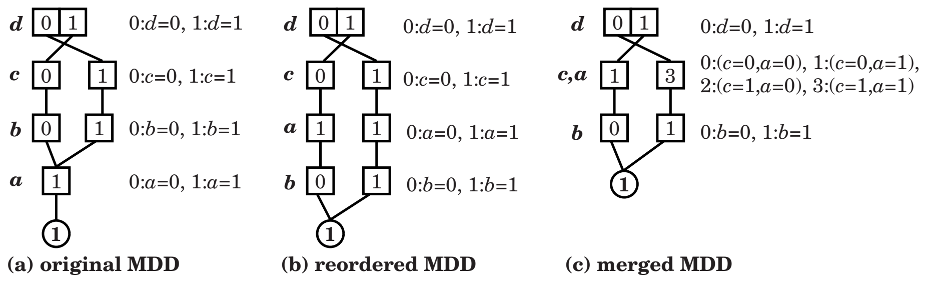


Fig. 3. An example where moving support variables closer increases the MDD size

```
Partition InvariantBasedMerging (Invariants $\mathbf{W}_{\mathcal{W},\mathcal{P}}$)
 1 $K \leftarrow |\mathcal{P}|$;
 2 $\pi \leftarrow (\{p_K\}, ..., \{p_1\})$;        • Initialize the partition with one place per class
 3 repeat
 4    for $m = K-1$ to 1 do
 5       $k = LevelToMerge(K, m, \pi, \mathbf{W}_{\mathcal{W},\mathcal{P}})$;
 6       if $k > m$ then
 7          $\pi \leftarrow (\pi_K, ..., \pi_{k+1}, \pi_k \cup \pi_m, \pi_{k-1}, ..., \pi_{m+1}, \pi_{m-1}, ..., \pi_1)$;
 8          $K \leftarrow K - 1$;        • The partition is now one class smaller
 9 until $\pi$ does not change;
10 return $\pi$;
```

```
int LevelToMerge (int $L$, int $m$, Partition $(\mathcal{Q}_L, ..., \mathcal{Q}_1)$, Invariants $\mathbf{W}_{\mathcal{W},\mathcal{P}}$)
 1 foreach $k = L$ downto $m+1$ do
 2    $\mathcal{W}' \leftarrow \{v \in \mathcal{W} \mid Support(v) \subseteq \mathcal{Q}_m \cup \bigcup_{l=k}^{L} \mathcal{Q}_l\}$;
 3    if $|\mathcal{Q}_m| = Rank(\mathbf{W}_{\mathcal{W}',\mathcal{Q}_m})$ then
 4       return $k$;
 5 return $m$;
```

Fig. 4. A greedy algorithm to iteratively merge MDD variables using Thm. 3

Proof. The proof is a specialization of the one of Thm. 2, noting that, there, we used the mapping $x_{k,m} = x_k n_m + x_m$ for simplicity. In reality, any mapping where $x_{k,m}$ can uniquely encode any *reachable* combination of x_k and x_m may be used. This is necessary in practice when using dynamic MDDs, where the sets $\mathcal{S}_l$, i.e., the bounds on the net places, are not known a priori. □

Greedy algorithm to merge MDD variables. Based on Thm. 3, Fig. 4 illustrates a greedy algorithm to merge as many MDD variables as possible, given a set of place invariants, while guaranteeing that the number of nodes and non-zero edges can only decrease.

For a Petri net, procedure *InvariantBasedMerging* in Fig. 4 takes a set of linearly independent place invariants, in the form of a matrix $\mathbf{W}_{\mathcal{W},\mathcal{P}}$, as input and assumes one place per variable in the initial MDD variable order (line 2). The procedure then traverses each level m of the MDD, from top to bottom according to the given partition π, and calls procedure *LevelToMerge* to compute the highest level k such that the m^{th} partition class π_m functionally depends on

$\mathcal{P}' = \bigcup_{K \geq l \geq k} \pi_l$. It does so by determining the set $\mathcal{W}'$ of invariants whose support is a subset of $\pi_m \cup \mathcal{P}'$, and by performing Gaussian elimination on submatrix $\mathbf{W}_{\mathcal{W}', \pi_m}$ to check whether it has full column rank (line 3 of *LevelToMerge*). If such level k exists, then π_m is merged with π_k, otherwise the partition remains unchanged. Procedure *InvariantBasedMerging* repeats this merging process until no more merging is possible, then it returns the final partition π.

The procedure has polynomial complexity, since it computes $O(|\mathcal{P}|^3)$ matrix ranks, in the worst case. In practice, due to the sparsity of matrix $\mathbf{W}$, the performance is excellent, as discussed in Sec. 4 We postpone to future work a discussion of whether it achieves the smallest possible number of MDD levels without increasing the number or size of the nodes according to Thm. 3.

3.2 Using Structural Invariants to Order State Variables

It is well-known that the variable order can greatly affect the efficiency of decision diagram algorithms, and that computing an optimal order is an NP-complete [2]. Thus, practical symbolic model-checking tools must rely on heuristics aimed at finding either a good order *statically*, i.e., prior to starting any symbolic manipulation, or at improving the order *dynamically*, i.e., during symbolic manipulation.

Focusing on static approaches, our locality-based encoding suggests that variable orders with small span $Top(t) - Bot(t) + 1$ for each transition t are preferable, both memory-wise when encoding $\mathcal{N}_t$, and time-wise when applying $\mathcal{N}_t$ to compute an image. Furthermore, since Saturation works on the nodes in a bottom-up fashion, it prefers orders where most spans are situated in lower levels. In the past, we have then considered the following static heuristics [39]:

- **SOS:** Minimize the sum of the transition spans, $\sum_{t \in \mathcal{T}} (Top(t) - Bot(t) + 1)$.
- **SOT:** Minimize the sum of the transition tops, $\sum_{t \in \mathcal{T}} Top(t)$.
- **Combined SOS/SOT:** Minimize $\sum_{t \in \mathcal{T}} Top(t)^{\alpha} \cdot (Top(t) - Bot(t) + 1)$.

The combined heuristic encompasses SOS and SOT, since the parameter α controls the relative importance of the size of the span vs. its location. When $\alpha = 0$, the heuristic becomes SOS, as it ignores the location of the span, while for $\alpha \gg 1$, it approaches SOT. For the test suite in [39], $\alpha = 1$ works generally well, confirming our intuition about the behavior of Saturation, namely that Saturation tends to perform better when both the size and the location of the spans is small.

We now propose to integrate the idea of an ordering heuristic based on transition locality with the equally intuitive idea that an order where variables in the support of an invariant are "close to each other," is preferable [29]. However, given the lesson of the previous section, we also wish to apply our greedy merging heuristic. There are four ways to approach this:

– For each possible permutation of the places, apply our merging heuristic. Then, evaluate the score of the chosen objective function (among the three above), and select the permutation that results in the minimum score. Of course, this results in the optimal order with respect to the chosen objective function, but the approach is not feasible except for very small nets.

- Choose one of the objective functions and, assuming one place per level, compute an order that produces a low score. Note that this is not necessarily the minimum score, as this is itself an NP-complete problem. Then, apply either our greedy merging heuristic, or a modified version of it that ensures that the achieved score is not worsened.
- Given an initial ordering of the places, use our greedy merging heuristic. Then, compute an order that produces a low, not necessarily minimal, score for the chosen objective function, subject to the constraints of Thm. 3, to keep the node size linear. For example, if $\mathbf{m}_a+\mathbf{m}_b+\mathbf{m}_c=N$ in every marking $\mathbf{m}$, if places a and b are not covered by any other invariant, if a and b have been merged together, and if they are at a level below that of c, then we cannot move them above c. If we did, a node encoding $\mathbf{m}_a$ and $\mathbf{m}_b$ could have $O(N^2)$ nonzero edges, since $\mathbf{m}_a+\mathbf{m}_b$ is not fixed until we know $\mathbf{m}_c$.
- Consider an invariant just like a transition, i.e., modify the chosen objective function to sum over both transitions and invariants, where the support of an invariant is treated just like the dependence list of a transition. Once the order is obtained, apply our greedy merging heuristic.

We adopt the last approach in conjunction with the SOT objective function, for several reasons. First, it is very similar in spirit to our original ordering approach, yet it adds novel information about invariants to guide the heuristic. Second, we have reasonably fast heuristics to solve SOT (indeed we even have a $\log n$ approximation algorithm for it), while the heuristics for SOS are not as fast, and those for the combined SOS/SOT problem are even slower. More importantly, when applying our greedy merging algorithm after the variable ordering heuristic, the span of an event is changed in unpredictable ways that do not preserve the optimality of the achieved score.

A fundamental observation is that, if place p is in the support of invariant v, any transition t that *modifies* p must also modify at least one other place q in the support of v. Thus, if p and p' are the lowest and highest places of the support of v according to the current MDD order, merging p with the second lowest place r in the support will not change the fact that p' is still the highest place in the support of v. Analogously, the highest place p'' determining $Top(t)$ is at least as high as q, which is at least as high as r, thus, again, p'' will still determine the value of $Top(t)$. Of course, the levels of p' and p'' are decreased by one, simply because the level of p, below them, is removed. Unfortunately, the same does not hold when p only *controls* the enabling or firing of a transition t, i.e., if there is an inhibitor arc from p to t or if p appears in the marking-dependent cardinality expression of arcs attached to t. In that case, merging p to a higher level k might increase the value of $Top(t)$ to k. Thus, for standard Petri nets with no inhibitor arcs and for the restricted self-modifying nets considered in [8], merging is guaranteed to improve the score of SOT, although it does not necessarily preserve optimality.

One danger or treating invariants like transitions in the scoring heuristic is that the number of invariants can be exponentially large, even when limiting ourselves to minimal ones (i.e., those whose support is not a superset of any other support). In such cases, the invariants would overwhelm the transitions and the resulting order would *de facto* be based almost exclusively on the invariants. To avoid this problem, we compute a set of linearly independent invariants and feed only those to our heuristic for SOT; clearly, this set will contain at most $|\mathcal{P}|$ elements, whence it is of the same order as $|\mathcal{T}|$ in practical models.

4 Experimental Results

We have implemented our static variable ordering merging ideas based on place invariants in the verification tool SMART [9], which supports Petri nets as front-end and reads an invariant matrix generated by the Petri-net tool GreatSPN [7]. This section reports our experimental results on a suite of asynchronous Petri net benchmarks for symbolic state-space generation.

We ran our experiments on a 3GHz Pentium workstation with 1GB RAM. Benchmarks *mmgt, dac, sentest, speed, dp, q, elevator,* and *key* are safe Petri nets taken from Corbett [16]. Benchmarks *knights* (board game model), *fms* and *kanban* [40] (manufacturing models), and *slot* [30], *courier* [42], and *ralep* [20] (protocol models) are Petri nets (without marking-dependent arcs, since GreatSPN does not accept this extension) from the SMART distribution.

Results. The first five columns of Table 1 show the model name and parameters, and the number of places ($\#P$), events ($\#T$) and place invariants computed by GreatSPN ($\#I$). The remaining columns are grouped according to whether the static variable order, computed via a fairly efficient logarithmic approximation for **SOT**, uses just the place-event matrix (**Event**) or the combined place-event+invariant matrix (**Event+Inv**). The approximation uses a randomized procedure, thus different parameters for the same model may result in different performance trends. For example, with **Event**, merging reduces the runtime of *courier* from 251 to 68sec when the parameter is 40, but has negligible effect when the parameter is 20.

The time for static variable ordering is stated in column **Time Ord**. For each group, we further report results according to whether variable merging is employed; method **No Merge** just uses the static order and therefore has one MDD variable per place of the Petri net, while **Merge** starts from the static order and merges variables using the proposed greedy algorithm of Fig. 4.

In addition, we state the run-time, peak, and final memory usage if the state-space generation with Saturation completes within 30 minutes. For **Merge**, we also report the number of MDD variables merged ($\#M$). The run-time spent for merging variables is not reported separately because it is quite small, always less than 5% of the total run-time, for any of the models. The time needed by GreatSPN to compute the invariants is shown in column **Time Inv**.

Discussion. From Table 1, we see the effectiveness of the new static variable ordering by comparing the two **No Merge** columns for **Event** and **Event+Inv**. The latter performs much better than the former on *mmgt*, *fms*, *slot*, *courier*, and *kanban*, slightly worse on *elevator* and *knights*, and similarly on the remaining

Table 1. Experimental results (Time in sec, Memory in KB;“>1800” means that run-time exceeds 1800 sec or memory exceeds 1GB)

Model	N	$\#P$	$\#T$	$\#I$	**Event**								**Event+Inv**							
					Time	**No Merge**			**Merge**				**Time**	**Time**	**No Merge**			**Merge**		
					Ord	**Time**	**Peak**	**Final**	**Time**	**Peak**	**Final**	$\#M$	**Inv**	**Ord**	**Time**	**Peak**	**Final**	**Time**	**Peak**	**Final**	$\#M$
mmgt	3	122	172	47	4.0	1.85	1575	83	1.78	1485	77	12	0.02	3.0	0.96	838	46	0.93	829	43	12
mmgt	4	158	232	48	6.0	18.05	20645	295	17.10	19294	280	14	0.02	5.0	4.71	5385	142	4.78	5375	132	14
dac	15	105	73	183	1.0	0.24	30	26	0.20	21	19	28	0.02	1.0	0.24	28	27	0.18	20	19	28
sentest	75	252	102	3315	2.0	0.55	49	44	0.25	20	17	157	1.07	2.0	0.53	49	45	0.23	20	17	157
sentest	100	327	127	5665	5.0	0.71	70	65	0.32	27	24	207	3.48	5.0	0.94	64	61	0.3	24	22	208
speed	1	29	31	10	0.0	0.07	48	6	0.06	32	4	10	0.01	0.0	0.09	44	6	0.07	29	4	10
dp	12	72	48	48	0.0	0.16	19	15	0.09	9	7	36	0.01	1.0	0.17	18	15	0.09	9	7	36
q	1	163	194	492	5.0	0.84	715	349	0.72	619	294	27	0.09	5.0	0.77	524	336	0.65	442	280	29
elevator	3	326	782	693	28.0	47.06	6570	1620	45.39	6532	1412	9	1.87	21.0	49.73	7403	1654	47.71	7359	1443	9
key	2	94	92	774	0.0	0.26	86	72	0.23	90	58	16	0.45	0.0	0.25	91	71	0.26	102	58	15
key	3	129	133	5491	3.0	0.54	231	161	0.53	210	145	18	127.11	2.0	0.51	211	146	0.46	196	136	18
knights	5	243	401	91	2.0	9.20	3321	60	7.03	2138	39	25	0.03	2.0	12.37	4084	60	9.5	2584	39	25
fms	20	38	20	27	0.0	2.58	1388	334	2.76	1371	317	3	0.01	0.0	0.39	189	66	0.5	180	57	3
fms	40	38	20	27	0.0	26.34	10480	1786	27.20	10418	1724	3	0.01	0.0	2.28	755	250	2.57	749	244	3
fms	80	38	20	27	0.0	93.59	19159	9068	110.20	18923	8831	3	0.01	0.0	31.16	9420	1383	32.7	9301	1263	3
slot	20	160	160	42	2.0	>1800	–	–	>1800	–	–	–	0.01	2.0	1.57	1658	122	1.35	1213	90	41
slot	40	320	320	82	12.0	>1800	–	–	>1800	–	–	–	0.03	8.0	10.96	11802	481	8.56	8540	353	81
courier	20	45	34	13	0.0	7.35	6985	871	7.20	6816	775	13	0.01	1.0	4.14	2693	267	3.89	2441	229	13
courier	40	45	34	13	0.0	251.06	108562	4260	68.22	39397	1126	13	0.01	1.0	25.38	12282	1127	24.99	11413	994	13
courier	80	45	34	13	1.0	>1800	–	–	>1800	–	–	–	0.01	1.0	191.34	57385	5902	187.28	50540	5212	13
kanban	20	16	16	6	0.0	307.66	51522	33866	192.56	44343	26687	4	0.01	0.0	0.93	513	55	1.23	443	45	4
kanban	40	16	16	6	0.0	539.11	134734	49478	402.62	113223	45753	4	0.01	0.0	7.61	3043	240	8.91	2777	206	4
kanban	80	16	16	6	0.0	>1800	–	–	>1800	–	–	–	0.01	0.0	85.02	20404	1249	99.07	19367	1124	4
ralep	7	91	140	21	2.0	22.73	25767	3424	20.64	21552	2526	21	0.01	3.0	23.00	27704	3349	22.82	24258	2613	21
ralep	8	104	168	24	2.0	99.79	90499	9166	106.03	80117	6572	24	0.01	6.0	85.20	88486	7872	81.5	66893	6006	24
ralep	9	117	198	27	2.0	359.68	238313	17531	429.62	223186	12605	27	0.01	3.0	361.09	232590	15463	387.61	196297	11891	27

benchmarks. The run-time for variable order computation is normally a small percentage of the run-times. The same can be said for invariant computation, with the exception of two models, *sentest* and *key*, where GreatSPN computes a large number of (non-minimal) invariants and requires more run-time than state-space generation itself (pathologically so for *key* with parameter 3). However, it must be stressed that the run-times for state-space generation are the ones obtained using our heuristic; if we were to use random or even just not as good orders, the state-space generation run-times would be much larger.

To see the effectiveness of invariants-based variable merging, one can compare the **No Merge** and **Merge** columns of Table 1, for either **Event** and **Event+Inv**. Merging almost always substantially improves the peak and final memory usage and results in comparable or better run-time performance, with up to a factor of three improvement for memory and time.

Even if merging is guaranteed to reduce the size of a given MDD, applying this idea is still a heuristic, as it changes the value of *Top* for the transition in the net, in such a way that Saturation may apply them in a different order that results in a larger peak size (in our benchmarks, this happens only for *key* with parameter 2). Overall, though, we believe that our ordering and merging heuristics can pave a way to a fully automated static ordering approach. This has a very practical impact, as it does not require a modeler to come up with a good order, and it reduces or eliminates altogether reliance on dynamic variable reordering, known to be quite expensive in practice.

5 Related Work

Most work on developing heuristics for finding good variable orders has been carried out in the context of digital-circuit verification and BDDs. Our focus in this paper is on *static* ordering, i.e., on finding a good ordering *before* constructing decision diagrams. In circuit verification, such approaches are typically based on a circuit's topological structure and work on the so-called *model connectivity graph*, by either searching, evaluating, or decomposing this graph. Grumberg, Livne, and Markovitch [18] present a good survey of these approaches and propose a static ordering based on "experience from training models". Dynamic grouping of boolean variables into MDD variables is proposed in [36]; however, invariants are not used to guide such grouping.

Our approach to static variable ordering with respect to Petri nets stands out as it considers *place invariants* and proposes *variable merging* instead of variable elimination. It must be pointed out that Pastor, Cortadella, and Roig mention in [29] that they "choose the ordering with some initial support from the structure of the Petri net (the P-invariants of the net)"; however, no details are given. More fundamentally, though, our work here shows that ordering using invariants is simply not as effective as ordering *and* merging using invariants.

Invariants are one popular approach to analyzing Petri nets [32,34]. With few exceptions, e.g., work by Schmidt [35] that utilizes transition invariants and research by Silva and his group [5] on performance throughput bounds, most researchers focus on place invariants. On the one hand, place invariants can help in identifying conservative upper bounds of a Petri net's reachable markings. Indeed, place invariants provide necessary, but not sufficient conditions on the reachability of a given marking. This in turn can benefit state-space generation algorithms, as is demonstrated, e.g., by Pastor, Cortadella, and Peña in [28].

On the other hand, place invariants can be used to reduce the amount of memory needed for storing a single marking [6,35], by exploiting the functional dependencies described by each invariant. When storing sets of markings with decision diagrams, this eliminates some decision-diagram variables. To determine which exact places or variables should be dropped, Davies, Knottenbelt, and Kritzinger present a heuristic in [17]. In that paper they also propose an ad-hoc heuristic for the static variable ordering within BDDs, based on finding pairs of similar subnets and interleaving the corresponding places' bit-vectors.

For the sake of completeness we note that general functional dependencies have also been studied by Hu and Dill [19]. In contrast to work in Petri nets where generated invariants are known to be correct, Hu and Dill do not assume the correctness of given functional dependencies, but prove them correct alongside verification. Last, but not least, we shall mention the approach to static variable ordering taken by Semenov and Yakovlev [37], who suggest to find a "close to optimal ordering" via net unfolding techniques.

6 Conclusions and Future Work

This paper demonstrated the importance of considering place invariants of Petri nets when statically ordering variables for symbolic state-space generation. Previous work focused either solely on optimizing event locality [39], or on eliminating variables based on invariance information [17]. The novel heuristic proposed in this paper enhances the former work by exploiting place invariants for *merging* variables, instead of eliminating them as is done in all related research. While merging is not an option for BDDs, it is suitable for MDD-based approaches, including our Saturation algorithm [10]. We proved that merging MDD variables always reduces MDD sizes, while eliminating variables may actually enlarge MDDs. In addition, for standard Petri nets, merging never breaks event locality and often improves it, thus benefiting Saturation.

The benchmarking conducted by us within SMART [9] showed that our heuristic outperforms related static variable-ordering approaches in terms of time-efficiency *and* memory-efficiency. Most importantly, this is the case for practical examples, such as large instances of the slotted-ring network and the kanban system which had been out of reach of existing state-space exploration technology before. Hence, using invariants in variable-ordering heuristics is crucial, but it must be done correctly. In particular, the widespread practice of eliminating variables based on invariance information is counter-productive and should be abandoned in favor of merging variables.

Future work should proceed along two directions. On the one hand, we wish to explore whether our greedy merging algorithm is optimal, in the sense that it reduces an MDD to the smallest number of MDD variables according to our merging rule. On the other hand, we intend to investigate whether place invariants are also beneficial in the context of *dynamic* variable ordering.

References

1. Aziz, A., Taşiran, S., Brayton, R.K.: BDD variable ordering for interacting finite state machines. In: DAC, pp. 283–288. ACM Press, New York (1994)
2. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. on Computers 45(9), 993–1002 (1996)
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comp. 35(8), 677–691 (1986)
4. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transition relations. In: Very Large Scale Integration, IFIP Transactions, North-Holland, pp. 49–58 (1991)
5. Campos, J., Chiola, G., Silva, M.: Ergodicity and throughput bounds of Petri nets with unique consistent firing count vectors. IEEE Trans. Softw. Eng. 17(2), 117–126 (1991)
6. Chiola, G.: Compiling techniques for the analysis of stochastic Petri nets. In: Modeling Techniques and Tools for Computer Performance Evaluation, pp. 11–24. Plenum Press, New York (1989)
7. Chiola, G., Franceschinis, G., Gaeta, R., Ribaudo, M.: GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. Performance Evaluation 24(1,2), 47–68 (1995)
8. Ciardo, G.: Petri nets with marking-dependent arc multiplicity: Properties and analysis. In: Valette, R. (ed.) Application and Theory of Petri Nets 1994. LNCS, vol. 815, pp. 179–198. Springer, Heidelberg (1994)
9. Ciardo, G., Jones, R.L., Miner, A.S., Siminiceanu, R.: Logical and stochastic modeling with SMART. Performance Evaluation 63, 578–608 (2006)
10. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state space generation. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 328–342. Springer, Heidelberg (2001)
11. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer, Heidelberg (2003)
12. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The Saturation algorithm for symbolic state space exploration. STTT 8(1), 4–25 (2006)
13. Ciardo, G., Yu, A.J.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 146–161. Springer, Heidelberg (2005)
14. Colom, J., Silva, M.: Improving the linearly based characterization of P/T nets. In: Advances in Petri Nets. LNCS, vol. 483, pp. 113–145. Springer, Heidelberg (1991)
15. Colom, J.M., Silva, M.: Convex geometry and semiflows in P/T nets: A comparative study of algorithms for the computation of minimal p-semiflows. In: ICATPN, pp. 74–95 (1989)
16. Corbett, J.C.: Evaluating deadlock detection methods for concurrent software. IEEE Trans. Softw. Eng. 22(3), 161–180 (1996)
17. Davies, I., Knottenbelt, W., Kritzinger, P.S.: Symbolic methods for the state space exploration of GSPN models. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 188–199. Springer, Heidelberg (2002)

18. Grumberg, O., Livne, S., Markovitch, S.: Learning to order BDD variables in verification. J. Art. Int. Res. 18, 83–116 (2003)
19. Hu, A.J., Dill, D.L.: Reducing BDD size by exploiting functional dependencies. In: DAC, pp. 266–271. ACM Press, New York (1993)
20. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. In: Foundations of Computer Science, pp. 150–158. IEEE Press, New York (1981)
21. Jeong, S.-W., Plessier, B., Hachtel, G.D., Somenzi, F.: Variable ordering and selection for FSM traversal. In: ICCAD, pp. 476–479. ACM Press, New York (1991)
22. 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)
23. McMillan, K.: Symbolic Model Checking: An Approach to the State-Explosion Problem. PhD thesis, Carnegie-Mellon Univ. (1992)
24. Miner, A.S.: Implicit GSPN reachability set generation using decision diagrams. Performance Evaluation 56(1-4), 145–165 (2004)
25. Miner, A.S., Ciardo, G.: Efficient reachability set generation and storage using decision diagrams. In: Donatelli, S., Kleijn, J.H.C.M. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 6–25. Springer, Heidelberg (1999)
26. Murata, T.: Petri nets: properties, analysis and applications. Proc. of the IEEE 77(4), 541–579 (1989)
27. Pastor, E., Cortadella, J.: Efficient encoding schemes for symbolic analysis of Petri nets. In: DATE, pp. 790–795. IEEE Press, New York (1998)
28. Pastor, E., Cortadella, J., Peña, M.: Structural methods to improve the symbolic analysis of Petri nets. In: Donatelli, S., Kleijn, J.H.C.M. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 26–45. Springer, Heidelberg (1999)
29. Pastor, E., Cortadella, J., Roig, O.: Symbolic analysis of bounded Petri nets. IEEE Trans.Computers 50(5), 432–448 (2001)
30. Pastor, E., Roig, O., Cortadella, J., Badia, R.: Petri net analysis using boolean manipulation. In: Valette, R. (ed.) Application and Theory of Petri Nets 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994)
31. Ramachandran, P., Kamath, M.: On place invariant sets and the rank of the incidence matrix of Petri nets. In: Systems, Man, and Cybernetics, pp. 160–165. IEEE Press, New York (1998)
32. Reisig, W.: Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets. Springer, Heidelberg (1998)
33. Roig, O., Cortadella, J., Pastor, E.: Verification of asynchronous circuits by BDD-based model checking of Petri nets. In: DeMichelis, G., Díaz, M. (eds.) Application and Theory of Petri Nets 1995. LNCS, vol. 935, pp. 374–391. Springer, Heidelberg (1995)
34. Sankaranarayanan, S., Sipma, H., Manna, Z.: Petri net analysis using invariant generation. In: Verification: Theory and Practice. LNCS, vol. 2772, pp. 682–701. Springer, Heidelberg (2003)
35. Schmidt, K.: Using Petri net invariants in state space construction. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 473–488. Springer, Heidelberg (2003)
36. Schmiedle, F., Günther, W., Drechsler, R.: Dynamic Re-Encoding During MDD Minimization. In: ISMVL (2000)
37. Semenov, A., Yakovlev, A.: Combining partial orders and symbolic traversal for efficient verification of asynchronous circuits. Techn. Rep. CS-TR 501, Newcastle Univ. (1995)
38. Sieling, D., Wegener, I.: NC-algorithms for operations on binary decision diagrams. Parallel Processing Letters 3, 3–12 (1993)

39. Siminiceanu, R., Ciardo, G.: New metrics for static variable ordering in decision diagrams. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 90–104. Springer, Heidelberg (2006)
40. Tilgner, M., Takahashi, Y., Ciardo, G.: SNS 1.0: Synchronized Network Solver. In: Manufacturing and Petri Nets, pp. 215–234 (1996)
41. Valk, R.: Generalizations of Petri nets. In: Gruska, J., Chytil, M.P. (eds.) Mathematical Foundations of Computer Science 1981. LNCS, vol. 118, pp. 140–155. Springer, Heidelberg (1981)
42. Woodside, C.M., Li, Y.: Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In: PNPM, pp. 64–73 (1991)

Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems

Hartmut Ehrig, Kathrin Hoffmann*, Julia Padberg, Ulrike Prange, and Claudia Ermel

Institute for Software Technology and Theoretical Computer Science
Technical University of Berlin, Germany

Abstract. Reconfigurable place/transition systems are Petri nets with initial markings and a set of rules which allow the modification of the net during runtime in order to adapt the net to new requirements of the environment. In this paper we use transformation rules for place/transition systems in the sense of the double pushout approach for graph transformation. The main problem in this context is to analyze under which conditions net transformations and token firing can be executed in arbitrary order. This problem is solved in the main theorems of this paper. Reconfigurable place/transition systems then are applied in a mobile network scenario.

Keywords: integration of net theory and graph transformations, parallel and sequential independence of net transformations and token firing.

1 Introduction

In [23], the concept of reconfigurable place/transition (P/T) systems has been introduced that is most important to model changes of the net structure while the system is kept running. In detail, a reconfigurable P/T-system consists of a P/T-system and a set of rules, so that not only the follower marking can be computed but also the structure can be changed by rule application to obtain a new P/T-system that is more appropriate with respect to some requirements of the environment. Moreover these activities can be interleaved.

For rule-based transformations of P/T-systems we use the framework of net transformations [17, 18] that is inspired by graph transformation systems [34]. The basic idea behind net transformation is the stepwise development of P/T-systems by given rules. Think of these rules as replacement systems where the left-hand side is replaced by the right-hand side while preserving a context. Petri nets that can be changed, have become a significant topic in the recent years, as the adaption of a system to a changing environment gets more and more important. Application areas cover e.g. computer supported cooperative work,

* This work has been partly funded by the research project forMA$_l$NET (see tfs.cs.tu-berlin.de/formalnet/) of the German Research Council.

multi agent systems, dynamic process mining or mobile networks. Moreover, this approach increases the expressiveness of Petri nets and allows a formal description of dynamic changes.

In this paper we continue our work by analyzing under which conditions a firing step is independent of a rule-based transformation step. Independence conditions for two firing steps of P/T-systems, i.e. being conflict free, are well-known and closely related to local Church-Rosser properties for graph resp. net transformations (see [34, 17, 18]) that are valid in the case of parallel and sequential independence of rule-based transformations. In [17] conditions for two transformation steps are given in the framework of high-level replacement systems with applications to net transformations, so that these transformation steps applied to the same P/T-system can be executed in arbitrary order, leading to the same result. But up to now it is open under which conditions a net transformation step and a firing step are independent of each other. In more detail, we assume that a given P/T-system represents a certain system state. The next evolution step can be obtained not only by token firing, but also by the application of one of the rules available. Hence, the question arises, whether each of these evolution steps can be postponed after the realization of the other, yielding the same result. Analogously, we ask ourselves if they can be performed in a different order without changing the result.

In Section 2 we present an interesting application of our concept in the area of mobile ad-hoc networks. While Section 3 reviews the notions of reconfigurable nets and net transformations, in Section 4 our main theorems concerning the parallel and sequential independence of net transformation and token firing are achieved. In Section 5 we show how these concepts and results can be put into the more general framework of algebraic higher-order nets. Finally, we outline related work and some interesting aspects of future work in Section 6.

2 Mobile Network Scenario

In this section we will illustrate the main idea of reconfigurable P/T-systems in the area of a mobile scenario. This work is part of a collaboration with some research projects where the main focus is on an adaptive workflow management system for mobile ad-hoc networks, specifically targeted to emergency scenarios[1]. So, as a running example we use a scenario in archaeological disaster/recovery: after an earthquake, a team (led by a team leader) is equipped with mobile devices (laptops and PDAs) and sent to the affected area to evaluate the state of archaeological sites and the state of precarious buildings. The goal is to draw a situation map in order to schedule restructuring jobs. The team is considered as an overall mobile ad-hoc network in which the team leader's device coordinates the other team member devices by providing suitable information (e.g. maps, sensible objects, etc.) and assigning activities. A typical cooperative process to be enacted by a team is shown in Fig. 1 as P/T-system (PN_1, M_1), where we

[1] MOBIDIS - http://www.dis.uniroma1.it/pub/mecella/projects/MobiDIS, MAIS - http://www.mais-project.it, IST FP6 WORKPAD - http://www.workpad-project.eu/

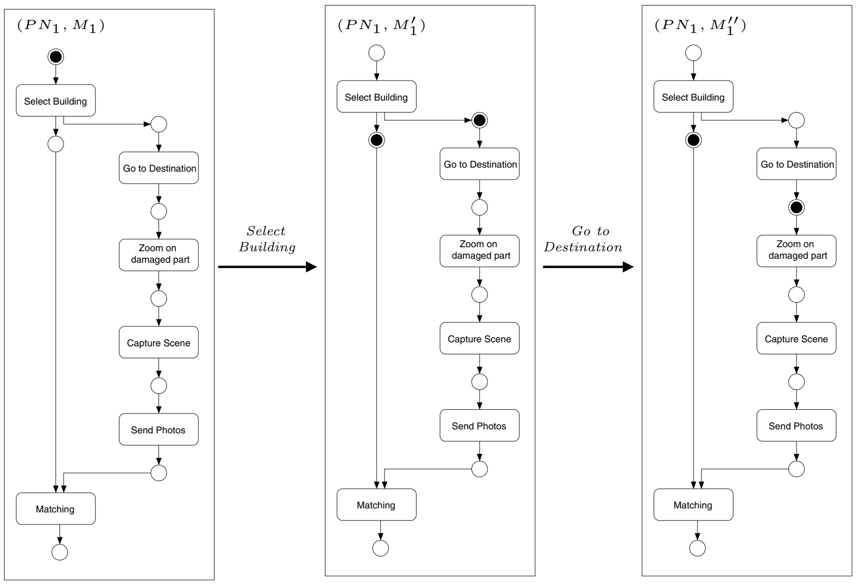


Fig. 1. Firing steps *Select Building* and *Go to Destination*

assume a team consisting of a team leader as picture store device and two team members as camera device and bridge device, respectively.

To start the activities of the camera device the follower marking of the P/T-system (PN_1, M_1) is computed by firing transition *Select Building* and we obtain the new P/T-system (PN_1, M_1') depicted in the middle of Fig. 1. In a next step the task *Go to Destination* can be executed (see right-hand side of Fig. 1).

To predict a situation of disconnection a movement activity of the bridge device has to be introduced in our system. In more detail, the workflow has to be extended by a task to follow the camera device. For this reason we provide the rule $prod_{follow}$ depicted in the upper row in Fig. 2. In general, a rule $prod = ((L, M_L) \xleftarrow{l} (K, M_K) \xrightarrow{r} (R, M_R))$ is given by three P/T-systems called left-hand side, interface, and right-hand side, respectively, and a span of two (specific) P/T-morphisms l and r. For the application of the rule $prod_{follow}$ to the P/T-system (PN_1, M_1) (see Fig. 1) we additionally need a match morphism m that identifies the relevant parts and has to respect the so-called gluing condition (see Section 3). Then the transformation step $(PN_1, M_1) \overset{prod_{follow}}{\Longrightarrow} (PN_2, M_2)$ as shown in Fig. 2 is given as follows: first, the transitions *Go to Destination* and *Send Photos* are deleted and we obtain the intermediate P/T-system (PN_0, M_0); then the transitions *Go to Destination*, *Send Photos* and *Follow Camera Device* together with their (new) environments are added. Note that a positive check of the gluing condition makes sure that the intermediate P/T-system is well-defined. Analogously, the application of the rule $prod_{follow}$ to the

P/T-system (PN_1, M_1') in the middle of Fig. 1 leads to the transformation step $(PN_1, M_1') \overset{prod_{follow}}{\Longrightarrow} (PN_2, M_2')$ in Fig. 3.

Note that in general token game and rule applications cannot be interleaved, e.g. if the transformation rule deletes the transition or a part of the marking used for the token firing. Thus we are looking for conditions such that firing steps and transformation steps can be performed in any order leading to the same P/T-system. In Section 4 we define in more detail conditions to ensure the independence of these activities.

Summarizing, our reconfigurable P/T-system $((PN_1, M_1), \{prod_{follow}\})$ consists of the P/T-system (PN_1, M_1) and the set of rules $\{prod_{follow}\}$ with one rule only. We can consider further rules, e.g. those given in [9,31], leading to a more complex reconfigurable P/T-system. But in this paper we use the simple reconfigurable P/T-system as an example to help the reader understand the main concepts.

3 Reconfigurable P/T-Systems

In this section we formalize reconfigurable P/T-systems. As net formalism we use P/T-systems following the notation of "Petri nets are Monoids" in [28]. In this notation a P/T-net is given by $PN = (P, T, pre, post)$ with pre- and post domain functions $pre, post : T \to P^{\oplus}$ and a P/T-system is given by (PN, M) with marking $M \in P^{\oplus}$, where $P^{\oplus}$ is the free commutative monoid over the set P of places with binary operation $\oplus$, e.g. the monoid notation $M = 2p_1 \oplus 3p_2$ means that we have two tokens on place p_1 and three tokens on p_2. Note that M can also be considered as function $M : P \to \mathbb{N}$ where only for a finite set $P' \subseteq P$ we have $M(p) \geq 1$ with $p \in P'$. We can switch between these notations by defining $\sum_{p \in P} M(p) \cdot p = M \in P^{\oplus}$. Moreover, for $M_1, M_2 \in P^{\oplus}$ we have $M_1 \leq M_2$ if $M_1(p) \leq M_2(p)$ for all $p \in P$. A transition $t \in T$ is M-enabled for a marking $M \in P^{\oplus}$ if we have $pre(t) \leq M$, and in this case the follower marking M' is given by $M' = M \ominus pre(t) \oplus post(t)$ and $(PN, M) \xrightarrow{t} (PN, M')$ is called firing step. Note that the inverse $\ominus$ of $\oplus$ is only defined in $M_1 \ominus M_2$ if we have $M_2 \leq M_1$.

In order to define rules and transformations of P/T-systems we introduce P/T-morphisms which preserve firing steps by Condition (1) below. Additionally they require that the initial marking at corresponding places is increasing (Condition (2)) or even stronger (Condition (3)).

Definition 1 (P/T-Morphisms)
Given P/T-systems $PN_i = (PN_i, M_i)$ with $PN_i = (P_i, T_i, pre_i, post_i)$ for $i = 1, 2$, a P/T-morphism $f : (PN_1, M_1) \to (PN_2, M_2)$ is given by $f = (f_P, f_T)$ with functions $f_P : P_1 \to P_2$ and $f_T : T_1 \to T_2$ satisfying

(1) $f_P^{\oplus} \circ pre_1 = pre_2 \circ f_T$ *and* $f_P^{\oplus} \circ post_1 = post_2 \circ f_T$ *and*
(2) $M_1(p) \leq M_2(f_P(p))$ *for all* $p \in P_1$.

Note that the extension $f_P^{\oplus} : P_1^{\oplus} \to P_2^{\oplus}$ of $f_P : P_1 \to P_2$ is defined by $f_P^{\oplus}(\sum_{i=1}^{n} k_i \cdot p_i) = \sum_{i=1}^{n} k_i \cdot f_P(p_i)$. (1) means that f is compatible with pre- and

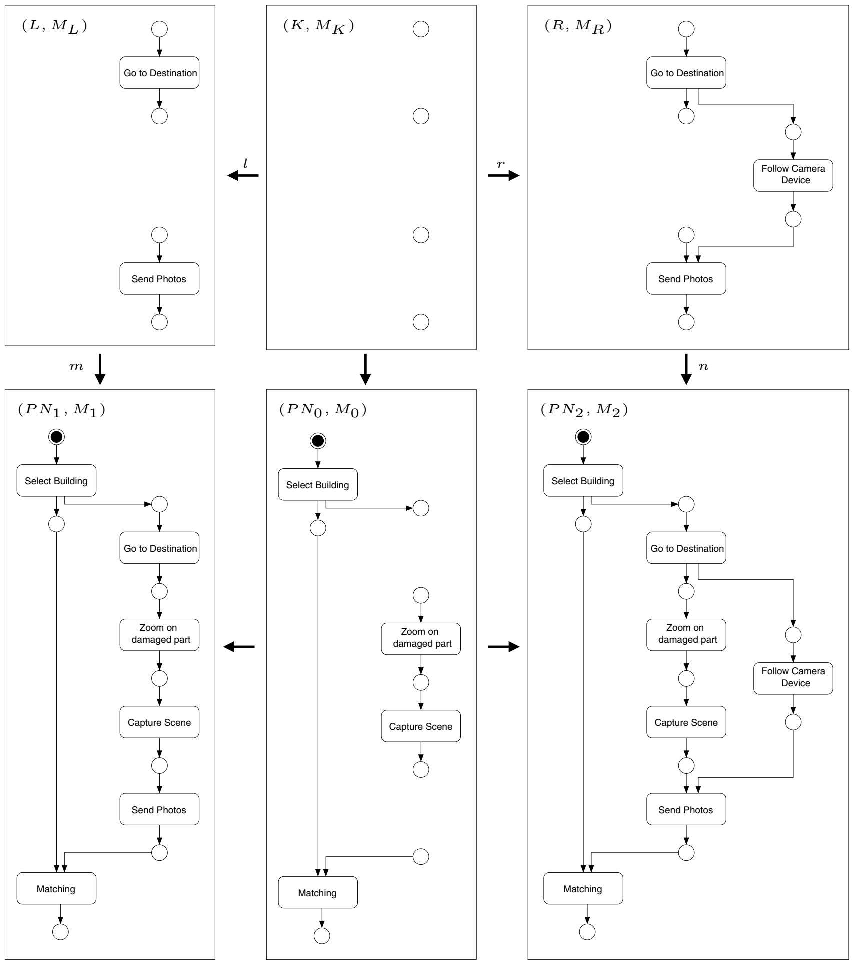


Fig. 2. Transformation step $(PN_1, M_1) \overset{prod_{follow}}{\Longrightarrow} (PN_2, M_2)$

post domain, and (2) that the initial marking of PN_1 at place p is smaller or equal to that of PN_2 at $f_P(p)$.

Moreover the P/T-morphism f is called strict if f_P and f_T are injective and

$$\textbf{(3)}\ M_1(p) = M_2(f_P(p)) \text{ for all } p \in P_1.$$

The category defined by P/T-systems and P/T-morphisms is denoted by **PTSys** *where the composition of P/T-morphisms is defined componentwise for places and transitions.*

Remark 1. For our morphisms we do not always have $f_P^{\oplus}(M_1) \leq M_2$. E.g. $M_1 = p_1 \oplus p_2, M_2 = p$ and $f_P(p_1) = f_P(p_2) = p$ implies $f_P^{\oplus}(M_1) = 2p > p = M_2$, but $M_1(p_1) = M_1(p_2) = 1 = M_2(p)$.

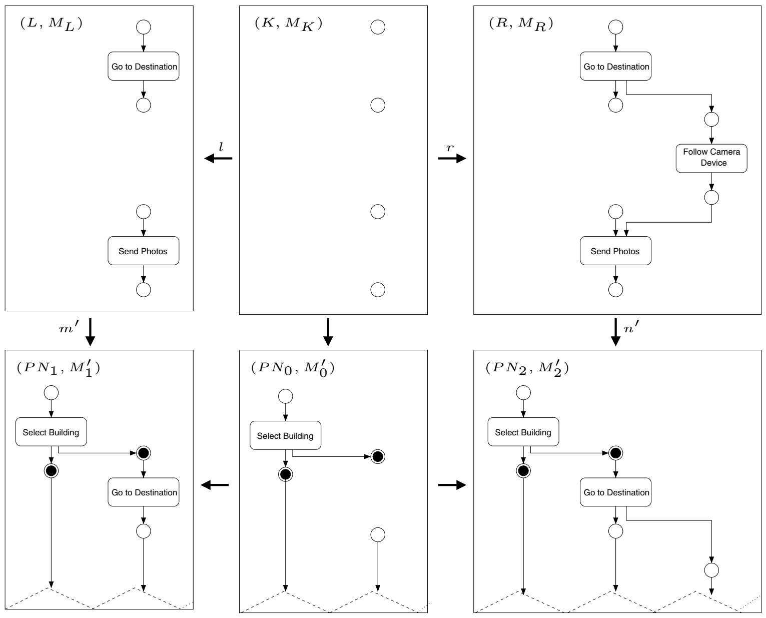


Fig. 3. Transformation step $(PN_1, M_1') \overset{prod_{follow}}{\Longrightarrow} (PN_2, M_2')$

As discussed in our paper [23] we are able to define the gluing of P/T-systems via P/T-morphisms by pushouts in the category **PTSys**. Informally, a pushout in a category **CAT** is a gluing construction of two objects over a specific interface. Especially we are interested in pushouts of the form where l is a strict and c is a general morphism. So, we can apply rules. Vice versa, given the left-hand side of a rule $(K, M_K) \xrightarrow{l} (L, M_L)$ (see Def. 3) and a match m: $(L, M_L) \to (PN_1, M_1)$ we have to construct a P/T-system (PN_0, M_0) such that (1) becomes a pushout. This construction requires the following gluing condition which has to be satisfied in order to apply a rule at a given match. The characterization of specific points is a sufficient condition for the existence and uniqueness of the so called pushout complement (PN_0, M_0), because it allows checking for the applicability of a rule to a given match.

$$\begin{tikzcd}
(K, M_K) \arrow[r, "l"] \arrow[d, "c"'] \arrow[dr, phantom, "(1)"] & (L, M_L) \arrow[d, "m"] \\
(PN_0, M_0) \arrow[r] & (PN_1, M_1)
\end{tikzcd}$$

Definition 2 (Gluing Condition for P/T-Systems)
Let $(L, M_L) \xrightarrow{m} (PN_1, M_1)$ be a P/T-morphism and $(K, M_K) \xrightarrow{l} (L, M_L)$ a strict morphism , then the gluing points GP, dangling points DP and the identification points IP of L are defined by

$$\begin{aligned}
GP &= l(P_K \cup T_K)\\
DP &= \{p \in P_L | \exists t \in (T_1 \setminus m_T(T_L)) : m_P(p) \in pre_1(t) \oplus post_1(t)\}\\
IP &= \{p \in P_L | \exists p' \in P_L : p \neq p' \wedge m_P(p) = m_P(p')\}\\
&\quad \cup \{t \in T_L | \exists t' \in T_L : t \neq t' \wedge m_T(t) = m_T(t')\}
\end{aligned}$$

A P/T-morphism $(L, M_L) \xrightarrow{m} (PN_1, M_1)$ and a strict morphism $(K, M_K) \xrightarrow{l} (L, M_L)$ satisfy the gluing condition, if all dangling and identification points are gluing points, i.e $DP \cup IP \subseteq GP$, and m is strict on places to be deleted, i.e.

$$\forall p \in P_L \setminus l(P_K) : M_L(p) = M_1(m(p)).$$

Example 1. In Section 2 examples of P/T-morphisms are given in Fig. 2 by $(K, M_K) \xrightarrow{l} (L, M_L)$ and $(L, M_L) \xrightarrow{m} (PN_1, M_1)$. For the dangling points we have $DP = P_L$ while the set of identification points IP is empty. So, these P/T-morphisms satisfy the gluing condition because the gluing points GP are also equal to the set of places P_L and all places are preserved.

Next we present rule-based transformations of P/T-systems following the double-pushout (DPO) approach of graph transformations in the sense of [34,17], which is restrictive concerning the treatment of unmatched transitions at places which should be deleted. Here the gluing condition forbids the application of rules in this case. Furthermore, items which are identified by a non injective match are both deleted or preserved by rule applications.

Definition 3 (P/T-System Rule)
A rule prod $= ((L, M_L) \xleftarrow{l} (K, M_K) \xrightarrow{r} (R, M_R))$ of P/T-systems consists of P/T-systems (L, M_L), (K, M_K), and (R, M_R), called left-hand side (LHS), interface, and right-hand side (RHS) of prod respectively, and two strict P/T-morphisms $(K, M_K) \xrightarrow{l} (L, M_L)$ and $(K, M_K) \xrightarrow{r} (R, M_R)$.

Note that we have not yet considered the firing of the rule nets (L, M_L), (K, M_K) and (R, M_R) as up to now no relevant use could be found. Nevertheless, from a theoretical point of view simultaneous firing of the nets (L, M_L), (K, M_K) and (R, M_R) is easy as the morphisms are marking strict. The firing of only one of these nets would require interesting extensions of the gluing condition.

Definition 4 (Applicability of Rules)
A rule prod $= ((L, M_L) \xleftarrow{l} (K, M_K) \xrightarrow{r} (R, M_R))$ is called applicable at the match $(L, M_L) \xrightarrow{m} (PN_1, M_1)$ if the gluing condition is satisfied for l and m. In this case we obtain a P/T-system (PN_0, M_0) leading to a net transformation step $(PN_1, M_1) \overset{prod,m}{\Longrightarrow} (PN_2, M_2)$ consisting of the following pushout diagrams (1) and (2). The P/T-morphism $n : (R, M_R) \to (PN_2, M_2)$ is called comatch of the transformation step.

$$\begin{tikzcd}
(L, M_L) \arrow[d, "m"'] \arrow[dr, phantom, "(\mathbf{1})"] & (K, M_K) \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "c"] \arrow[dr, phantom, "(\mathbf{2})"] & (R, M_R) \arrow[d, "n"] \\
(PN_1, M_1) & (PN_0, M_0) \arrow[l, "l^*"] \arrow[r, "r^*"'] & (PN_2, M_2)
\end{tikzcd}$$

Now we are able to define reconfigurable P/T-systems, which allow modifying the net structure using rules and net transformations of P/T-systems.

Definition 5 (Reconfigurable P/T-Systems)
Given a P/T-system (PN, M) and a set of rules $RULES$, a reconfigurable P/T-system is defined by $((PN, M), RULES)$.

Examples of rule applications and of a reconfigurable P/T-system can be found in Section 2.

4 Independence of Net Transformations and Token Firing

In this section we analyze under which conditions net transformations and token firing of a reconfigurable P/T-system as introduced in Section 3 can be executed in arbitrary order. These conditions are called (co-)parallel and sequential independence. Note that independence conditions for two firing steps of P/T-systems are well-known and independence of two transformation steps is analyzed already for high-level replacement systems with applications to Petri net transformations in [17]. We start with the situation where a transformation step and a firing step are applied to the same P/T-system. This leads to the notion of parallel independence.

Definition 6 (Parallel Independence)
A transformation step $(PN_1, M_1) \overset{prod,m}{\Longrightarrow} (PN_2, M_2)$ of P/T-systems and a firing step $(PN_1, M_1) \xrightarrow{t_1} (PN_1, M_1')$ for $t_1 \in T_1$ are called parallel independent if

(1) *t_1 is not deleted by the transformation step and*
(2) *$M_L(p) \leq M_1'(m(p))$ for all $p \in P_L$ with $(L, M_L) = LHS(prod)$.*

Parallel independence allows the execution of the transformation step and the firing step in arbitrary order leading to the same P/T-system.

Theorem 1 (Parallel Independence). *Given parallel independent steps $(PN_1, M_1) \overset{prod,m}{\Longrightarrow} (PN_2, M_2)$ and $(PN_1, M_1) \xrightarrow{t_1} (PN_1, M_1')$ with $t_1 \in T_1$ then there is a corresponding $t_2 \in T_2$ with firing step $(PN_2, M_2) \xrightarrow{t_2} (PN_2, M_2')$ and a transformation step $(PN_1, M_1') \overset{prod,m'}{\Longrightarrow} (PN_2, M_2')$ with the same marking M_2'.*

$$\begin{tikzcd}
 & (PN_1, M_1) \arrow[dl, Rightarrow, "prod{,}m"'] \arrow[dr, "t_1"] & \\
(PN_2, M_2) \arrow[dr, "t_2"'] & & (PN_1, M_1') \arrow[dl, Rightarrow, "prod{,}m'"] \\
 & (PN_2, M_2') &
\end{tikzcd}$$

Remark 2. Cond. (1) in Def. 6 is needed to fire t_2 in (PN_2, M_2), and Cond. (2) in Def. 6 is needed to have a valid match m' in (PN_1, M_1'). Note that $m'(x) = m(x)$ for all $x \in P_L \cup T_L$.

Proof. Parallel independence implies that $t_1 \in T_1$ is preserved by the transformation step $(PN_1, M_1) \overset{prod,m}{\Longrightarrow} (PN_2, M_2)$. Hence there is a unique $t_0 \in T_0$ with $l^*(t_0) = t_1$. Let $t_2 = r^*(t_0) \in T_2$ in the following pushouts (1) and (2), where l^* and r^* are strict.

$$\begin{tikzcd}
(L, M_L) \arrow[d, "m"'] \arrow[dr, phantom, "(\mathbf{1})"] & (K, M_K) \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(\mathbf{2})"] & (R, M_R) \arrow[d, "n"] \\
(PN_1, M_1) & (PN_0, M_0) \arrow[l, "l^*"] \arrow[r, "r^*"'] & (PN_2, M_2)
\end{tikzcd}$$

Now t_1 being enabled under M_1 in PN_1 implies $pre_1(t_1) \leq M_1$. Moreover, l^* and r^* strict implies $pre_0(t_0) \leq M_0$ and $pre_2(t_2) \leq M_2$. Hence t_2 is enabled under M_2 in PN_2 and we define $M_2' = M_2 \ominus pre_2(t_2) \oplus post_2(t_2)$.

Now we consider the second transformation step, with m' defined by $m'(x) = m(x)$ for $x \in P_L \cup T_L$.

$$\begin{tikzcd}
(L, M_L) \arrow[d, "m'"'] \arrow[dr, phantom, "(\mathbf{1'})"] & (K, M_K) \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(\mathbf{2'})"] & (R, M_R) \arrow[d, "n'"] \\
(PN_1, M_1') & (PN_0, M_0') \arrow[l, "l^{*\prime}"] \arrow[r, "r^{*\prime}"'] & (PN_2, M_2'')
\end{tikzcd}$$

m' is a P/T-morphism if for all $p \in P_L$ we have

(a) $M_L(p) \leq M_1'(m'(p))$,

and the match m' is applicable at M_1', if

(b) $IP \cup DP \subseteq GP$ and for all $p \in P_L \setminus l(P_K)$ we have $M_L(p) = M_1'(m(p))$ (see gluing condition in Def. 2).

Cond. (a) is given by Cond. (2) in Def. 6, because we assume that $(PN_1, M_1) \overset{prod,m}{\Longrightarrow} (PN_2, M_2)$ and $(PN_1, M_1) \xrightarrow{t_1} (PN_1, M_1')$ with $t_1 \in T_1$ are parallel independent. Moreover, the match m being applicable at M_1 implies $IP \cup DP \subseteq GP$ and for all $p \in P_L \setminus l(P_K)$ we have $M_L(p) = M_1(m(p)) = M_1'(m(p))$ by Lemma 1 below using the fact that there is a firing step $(PN_1, M_1) \xrightarrow{t_1} (PN_1, M_1')$. The application of *prod* along m' leads to the P/T-system (PN_2, M_2''), where $l^{*\prime}(x) = l^*(x)$, $r^{*\prime}(x) = r^*(x)$ for all $x \in P_0 \cup T_0$, and $n^{*\prime}(x) = n^*(x)$ for all $x \in P_R \cup T_R$.

Finally, it remains to show that $M_2' = M_2''$. By construction of the transformation steps $(PN_1, M_1) \overset{prod,m}{\Longrightarrow} (PN_2, M_2)$ and $(PN_1, M_1') \overset{prod,m'}{\Longrightarrow} (PN_2, M_2'')$ we have

(1) for all $p_0 \in P_0$: $M_2(r^*(p_0)) = M_0(p_0) = M_1(l^*(p_0))$,
(2) for all $p \in P_R \setminus r(P_K)$: $M_2(n(p)) = M_R(p)$,
(3) for all $p_0 \in P_0$: $M_2''(r^*(p_0)) = M_0'(p_0) = M_1'(l^*(p_0))$ and
(4) for all $p \in P_R \setminus r(P_K)$: $M_2''(n'(p)) = M_R(p)$.

By construction of the firing steps $(PN_1, M_1) \xrightarrow{t_1} (PN_1, M_1')$ and $(PN_2, M_2) \xrightarrow{t_2} (PN_2, M_2')$ we have

(5) for all $p_1 \in P_1$: $M_1'(p_1) = M_1(p_1) \ominus \mathit{pre}_1(t_1)(p_1) \oplus \mathit{post}_1(t_1)(p_1)$ and
(6) for all $p_2 \in P_2$: $M_2'(p_2) = M_2(p_2) \ominus \mathit{pre}_2(t_2)(p_2) \oplus \mathit{post}_2(t_2)(p_2)$.

Moreover, l^* and r^* strict implies the injectivity of l^* and r^* and we have

(7) for all $p_0 \in P_0$: $\mathit{pre}_0(t_0)(p_0) = \mathit{pre}_1(t_1)(l^*(p_0)) = \mathit{pre}_2(t_2)(r^*(p_0))$ and
$\mathit{post}_0(t_0)(p_0) = \mathit{post}_1(t_1)(l^*(p_0)) = \mathit{post}_2(t_2)(r^*(p_0))$.

To show that this implies

(8) $M_2' = M_2''$,

it is sufficient to show

(8a) for all $p \in P_R \setminus r(P_K)$: $M_2''(n'(p)) = M_2'(n(p))$ and
(8b) for all $p_0 \in P_0$: $M_2''(r^*(p_0)) = M_2'(r^*(p_0))$.

First we show that condition (8a) is satisfied. For all $p \in P_R \setminus r(P_K)$ we have

$$M_2''(n'(p)) \overset{(4)}{=} M_R(p) \overset{(2)}{=} M_2(n(p)) \overset{(6)}{=} M_2'(n(p))$$

because $n(p)$ is neither in the pre domain nor in the post domain of t_2, which are in $r^*(P_0)$ because t_2 is not created by the rule (see Lemma 1, applied to the inverse rule prod^{-1}).

Next we show that condition (8b) is satisfied. For all $p_0 \in P_0$ we have

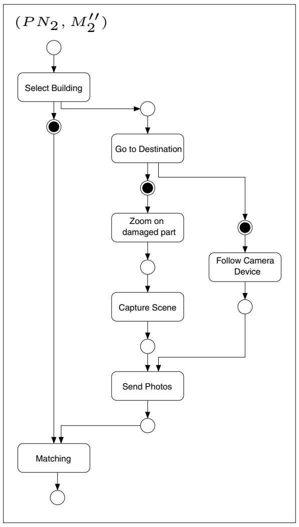
$$\begin{aligned}
M_2''(r^*(p_0)) &\overset{(3)}{=} M_0'(p_0) \\
&\overset{(3)}{=} M_1'(l^*(p_0)) \\
&\overset{(5)}{=} M_1(l^*(p_0)) \ominus \mathit{pre}_1(t_1)(l^*(p_0)) \oplus \mathit{post}_1(t_1)(l^*(p_0)) \\
&\overset{(1)\text{ and }(7)}{=} M_2(r^*(p_0)) \ominus \mathit{pre}_2(t_2)(r^*(p_0)) \oplus \mathit{post}_2(t_2)(r^*(p_0)) \\
&\overset{(6)}{=} M_2'(r^*(p_0))
\end{aligned}$$

It remains to show Lemma 1 which is used in the proof of Theorem 1.

Lemma 1. *For all $p \in P_L \setminus l(P_K)$ we have $m(p) \notin \mathit{dom}(t_1)$, where $\mathit{dom}(t_1)$ is union of pre- and post domain of t_1, and t_1 is not deleted.*

Proof. Assume $m(p) \in \mathit{dom}(t_1)$.

Case 1 ($t_1 = m(t)$ for $t \in T_L$): t_1 not being deleted implies $t \in l(T_K)$. Hence there exists $p' \in \mathit{dom}(t) \subseteq l(P_K)$, such that $m(p') = m(p)$; but this is a contradiction to $p \in P_L \setminus l(P_K)$ and the fact that m cannot identify elements of $l(P_K)$ and $P_L \setminus l(P_K)$.



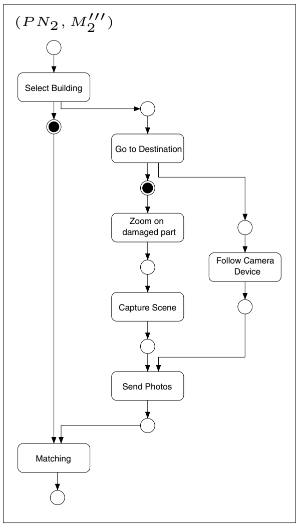


Fig. 4. P/T-systems (PN_2, M_2'') and (PN_2, M_2''')

Case 2 ($t_1 \notin m(T_L)$): $m(p) \in dom(t_1)$ implies by the gluing condition in Def. 2, that $p \in l(P_K)$, but this is a contradiction to $p \in P_L \setminus l(P_K)$.

Example 2. The firing step $(PN_1, M_1) \xrightarrow{Select\ Building} (PN_1, M_1')$ (see Fig. 1) and the transformation step $(PN_1, M_1) \overset{prod_{follow}}{\Longrightarrow} (PN_2, M_2)$ (see Fig. 2) are parallel independent because the transition *Select Building* is not deleted by the transformation step and the marking M_L is empty. Thus, the firing step can be postponed after the transformation step or, vice versa, the rule $prod_{follow}$ can be applied after token firing yielding the same result (PN_2, M_2') in Fig. 5.

In contrast the firing step $(PN_1, M_1') \xrightarrow{Go\ to\ Destination} (PN_1, M_1'')$ (see Fig. 1) and the transformation step $(PN_1, M_1') \overset{prod_{follow}}{\Longrightarrow} (PN_2, M_2')$ (see Fig. 3) are not parallel independent because the transition *Go to Destination* is deleted and afterwards reconstructed by the transformation step (it is not included in the interface K). In fact, the new transition *Go to Destination* in (PN_2, M_2') could be fired leading to (PN_2, M_2'') (see Fig. 4) and vice versa we could fire *Go to Destination* in (PN_1, M_1') and then apply $prod_{follow}$ leading to (PN_2, M_2''') (see Fig. 4), but we would have $M_2'' \neq M_2'''$.

In the first diagram in Theorem 1 we have required that the upper pair of steps is parallel independent leading to the lower pair of steps. Now we consider the situations that the left, right or lower pair of steps are given - with a suitable notion of independence - such that the right, left and upper part of steps can be constructed, respectively.

Definition 7 (Sequential and Coparallel Independence). *In the following diagram with $LHS(prod) = (L, M_L)$, $RHS(prod) = (R, M_R)$, m and m' are matches and n and n' are comatches of the transformation steps with $m(x) = m'(x)$ for $x \in P_L \cup T_L$ and $n(x) = n'(x)$ for $x \in P_R \cup T_R$, we say that*

$$\begin{tikzcd}
 & (PN_1, M_1) \arrow[dl, Rightarrow, "{(prod,m,n)}"'] \arrow[dr, "t_1"] & \\
(PN_2, M_2) \arrow[dr, "t_2"'] & & (PN_1, M_1') \arrow[dl, Rightarrow, "{(prod,m',n')}"] \\
 & (PN_2, M_2') &
\end{tikzcd}$$

1. *the left pair of steps, short $((prod, m, n), t_2)$, is sequentially independent if*
   (a) *t_2 is not created by the transformation step*
   (b) *$M_R(p) \leq M_2'(n(p))$ for all $p \in P_R$*
2. *the right pair of steps, short $(t_1, (prod, m', n'))$, is sequentially independent if*
   (a) *t_1 is not deleted by the transformation step*
   (b) *$M_L(p) \leq M_1(m'(p))$ for all $p \in P_L$*
3. *the lower pair of steps, short $(t_2, (prod, m', n'))$, is coparallel independent if*
   (a) *t_2 is not created by the transformation step*
   (b) *$M_R(p) \leq M_2(n'(p))$ for all $p \in P_R$*

Example 3. The pair of steps (*Select Building*, $(prod_{follow}, m', n')$) depicted in Fig. 5 is sequentially independent because the transition *Select Building* is not deleted by the transformation step and the marking M_L is empty. Analogously, the pair of steps ($(prod_{follow}, m, n)$, *Select Building*) depicted in Fig. 6 is sequentially independent because the transition *Select Building* is not created by the transformation step and the marking M_R is empty. For the same reason the pair (*Select Building*,$(prod_{follow}, m', n')$) is coparallel independent.

Remark 3. Note that for $prod = ((L, M_L) \xleftarrow{l} (K, M_K) \xrightarrow{r} (R, M_R))$ we have $prod^{-1} = ((R, M_R) \xleftarrow{r} (K, M_K) \xrightarrow{l} (L, M_L))$ and each direct transformation $(PN_1, M_1) \overset{prod}{\Longrightarrow} (PN_2, M_2)$ with match m, comatch n and pushout diagrams (1) and (2) as given in Def. 4 leads to a direct transformation $(PN_2, M_2) \overset{prod^{-1}}{\Longrightarrow} (PN_1, M_1)$ with match n and comatch m by interchanging pushout diagrams (1) and (2).

Given a firing step $(PN_1, M_1) \xrightarrow{t_1} (PN_1, M_1')$ with $M_1' = M_1 \ominus pre_1(t_1) \oplus post_1(t_1)$ we can formally define an inverse firing step $(PN_1, M_1') \xrightarrow{t_1^{-1}} (PN_1, M_1)$ with $M_1 = M_1' \ominus post_1(t_1) \oplus pre_1(t_1)$ if $post_1(t_1) \leq M_1'$, such that firing and inverse firing are inverse to each other.

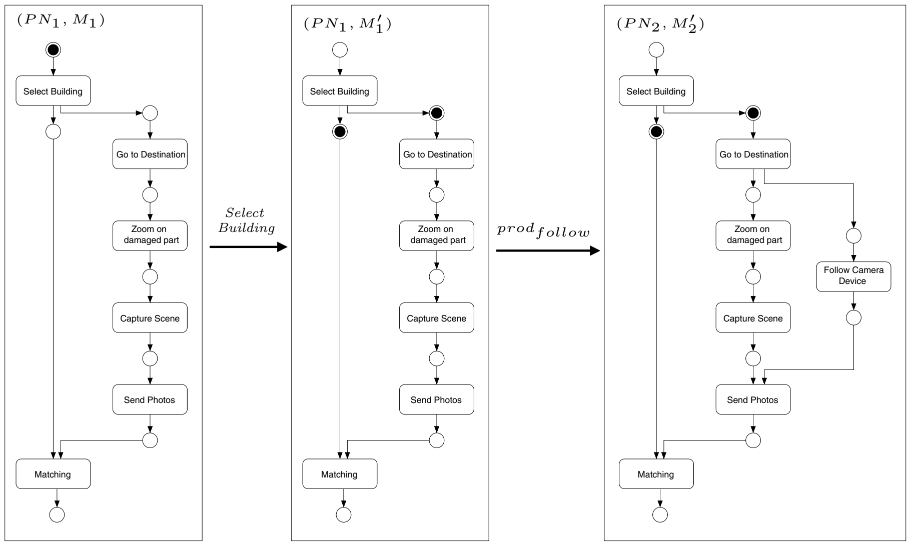


Fig. 5. Pair of steps (*Select Building*, $(prod_{follow}, m', n')$)

Formally all the notions of independence in Def. 7 can be traced back to parallel independence using inverse transformation steps based on $(prod^{-1}, n, m)$ and $(prod^{-1}, n', m')$ and inverse firing steps t_1^{-1} and t_2^{-1} in the following diagram.

$$\begin{tikzcd}
 & (PN_1, M_1) \arrow[dl, shift right, "(prod{,}m{,}n)"'] \arrow[dr, shift left, "t_1"] & \\
(PN_2, M_2) \arrow[ur, shift right, "(prod^{-1}{,}n{,}m)"'] \arrow[dr, shift left, "t_2"] & & (PN_1, M_1') \arrow[ul, shift left, "t_1^{-1}"] \arrow[dl, shift right, "(prod{,}m'{,}n')"'] \\
 & (PN_2, M_2') \arrow[ul, shift left, "t_2^{-1}"] \arrow[ur, shift right, "(prod^{-1}{,}n'{,}m')"'] &
\end{tikzcd}$$

1. $((prod, m, n), t_2)$ is sequentially independent iff $((prod^{-1}, n, m), t_2)$ is parallel independent.
2. $(t_1, (prod, m', n'))$ is sequentially independent iff $((prod, m', n'), t_1^{-1})$ is parallel independent.
3. $(t_2, (prod, m', n'))$ is coparallel independent iff $((prod^{-1}, n', m'), t_2^{-1})$ is parallel independent.

Now we are able to extend Theorem 1 on parallel independence showing that resulting steps in the first diagram of Theorem 1 are sequentially and coparallel independent.

Theorem 2 (Parallel and Sequential Independence). *In Theorem 1, where we start with parallel independence of the upper steps in the following diagram with* *match m and comatch n, we have in addition the following sequential and coparallel independence in the following diagram:*

$$\begin{tikzcd}
 & (PN_1, M_1) \arrow[dl, Rightarrow, "(prod,m,n)"'] \arrow[dr, "t_1"] & \\
(PN_2, M_2) \arrow[dr, "t_2"'] & & (PN_1, M_1') \arrow[dl, Rightarrow, "(prod,m',n')"] \\
 & (PN_2, M_2') &
\end{tikzcd}$$

1. *The left pair of steps, short $((prod, m, n), t_2)$, is sequentially independent.*
2. *The right pair of steps, short $(t_1, (prod, m', n'))$, is sequentially independent.*
3. *The lower pair of steps, short $(t_2, (prod, m', n'))$, is coparallel independent.*

Proof. We use the proof of Theorem 1.

1. (a) t_2 is not created because it corresponds to $t_1 \in T_1$ which is not deleted.
   (b) We have $M_R(p) \leq M_2'(n(p))$ for all $p \in P_R$ by construction of the pushout $(2')$ with $M_2'' = M_2'$.
2. (a) t_1 is not deleted by the assumption of parallel independence.
   (b) $M_L(p) \leq M_1(m(p))$ for all $p \in P_L$ by pushout (1).
3. (a) t_2 is not created as shown in the proof of 1. (a).
   (b) $M_R(p) \leq M_2(n(p))$ for all $p \in P_R$ by pushout (2).

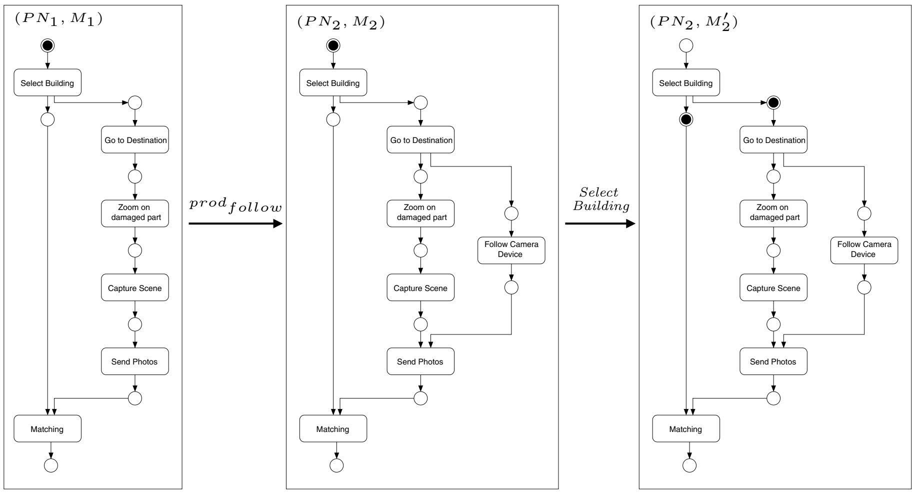


Fig. 6. Pair of steps $((prod_{follow}, m, n)$, *Select Building*)

In Theorem 2 we have shown that parallel independence implies sequential and coparallel independence. Now we show vice versa that sequential (coparallel) independence implies parallel and coparallel (parallel and sequential) independence.

Theorem 3 (Sequential and (Co-)Parallel Independence)

1. *Given the left sequentially independent steps in diagram (1) then also the right steps exist, s.t. the upper (right, lower) pair is parallel (sequentially, coparallel) independent.*
2. *Given the right sequentially independent steps in diagram (1) then also the left steps exist, s.t. the upper (left, lower) pair is parallel (sequentially, coparallel) independent.*
3. *Given the lower coparallel independent steps in diagram (1) then also the upper steps exist, s.t. the upper (left,right) pair is parallel (sequentially, sequentially) independent.*

$$\begin{tikzcd}
 & (PN_1, M_1) \arrow[dl, Rightarrow, "(prod{,}m{,}n)"'] \arrow[dr, "t_1"] & \\
(PN_2, M_2) \arrow[dr, "t_2"'] & (1) & (PN_1, M_1') \arrow[dl, Rightarrow, "(prod{,}m'{,}n')"] \\
 & (PN_2, M_2') &
\end{tikzcd}$$

Proof 1. Using Remark 3, left sequential independence in (1) corresponds to parallel independence in (2). Applying Theorem 1 and Theorem 2 to the left pair in (2) we obtain the right pair such that the upper and lower pairs are sequentially and the right pair coparallel independent. This implies by Remark 3 that the upper (right, lower) pairs in (1) are parallel (sequentially, coparallel) independent.

$$\begin{tikzcd}
 & (PN_1, M_1) \arrow[dr, "t_1"] & \\
(PN_2, M_2) \arrow[ur, Rightarrow, "(prod^{-1}{,}n{,}m)"] \arrow[dr, "t_2"'] & (2) & (PN_1, M_1') \\
 & (PN_2, M_2') \arrow[ur, Rightarrow, "(prod^{-1}{,}n'{,}m')"'] &
\end{tikzcd}$$

The proofs of items *2.* and *3.* are analogous to the proof of *1.*

5 General Framework of Net Transformations

In [23], we have introduced the paradigm "nets and rules as tokens" using a high-level model with suitable data type part. This model called algebraic higher-order (AHO) system (instead of high-level net and replacement system as in [23]) exploits some form of control not only on rule application but also on token firing. In general an AHO-system is defined by an algebraic high-level net with system places and rule places as for example shown in Fig. 7, where the marking is given by a suitable P/T-system resp. rule on these places. For a detailed description of the data type part, i.e. the AHO-SYSTEM-signature and corresponding algebra A, we refer to [23].

In the following we review the behavior of AHO-systems according to [23]. With the symbol $Var(t)$ we indicate the set of variables of a transition t, i.e., the set of all variables occurring in pre- and post domain and in the firing-condition of t. The marking M determines the distribution of P/T-systems and rules in an AHO-system, which are elements of a given higher-order algebra A. Intuitively, P/T-systems and rules can be moved along AHO-system arcs and can be modified during the firing of transitions. The follower marking is computed by the evaluation of net inscriptions in a variable assignment $v : Var(t) \to A$. The transition t is enabled in a marking M, if and only if (t, v) is consistent, that is if the evaluation of the firing condition is fulfilled. Then the follower marking after firing of transition t is defined by removing tokens corresponding to the net inscription in the pre domain of t and adding tokens corresponding to the net inscription in the post domain of t.

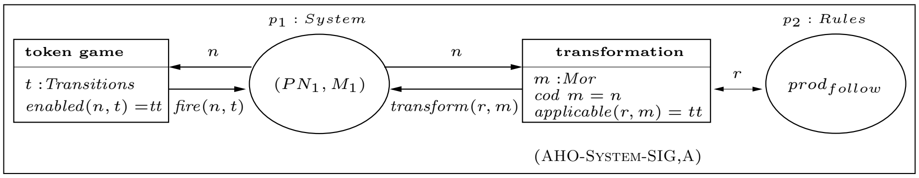


Fig. 7. Algebraic higher-order system

The transitions in the AHO-system in Fig. 7 realize on the one hand firing steps and on the other hand transformation steps as indicated by the net inscriptions $fire(n,t)$ and $transform(r,m)$, respectively. The initial marking is the reconfigurable P/T-system given in Section 2, i.e. the P/T-system (PN_1, M_1) given in Fig. 1 is on the place p_1, while the marking of the place p_2 is given by the rule $prod_{follow}$ given in Fig. 2. To compute the follower marking of the P/T-system we use the transition *token game* of the AHO-system. First the variable n is assigned to the P/T-system (PN_1, M_1) and the variable t to the transition *Select Building*. Because this transition is enabled in the P/T-system, the firing condition is fulfilled. Finally, due to the evaluation of the term $fire(n,t)$ we obtain the new P/T-system (PN_1, M_1') (see Fig. 1).

For changing the structure of P/T-systems the transition *transformation* is provided in Fig. 7. Again, we have to give an assignment v for the variables of the transition, i.e. variables n, m and r, where $v(n) = (PN_1, M_1)$, $v(m)$ is a suitable match morphism and $v(r) = prod_{follow}$. The firing condition *cod* $m = n$ ensures that the codomain of the match morphism is equal to (PN_1, M_1), while the second condition *applicable*(r, m) checks the gluing condition, i.e. if the rule $prod_{follow}$ is applicable with match m. Afterwards, the transformation step depicted in Fig. 2 is computed by the evaluation of the net inscription $transform(r, m)$ and the effect of firing the transition *transformation* is the removal of the P/T-system (PN_1, M_1) from place p_1 in Fig. 7 and adding the P/T-system (PN_2, M_2) to it. The pair (or sequence) of firing and transformation steps discussed in the last sections is reflected by firing of the transitions one after the other in our AHO-system. Thus, the results presented in this paper are most important for the analysis of AHO-systems.

6 Conclusion

This paper continues our work on "nets and rules as tokens" [23] by transferring the results of local Church-Rosser, which are well known for term rewriting and graph transformations, to the consecutive evolution of a P/T-system by token firing and rule applications. We have presented conditions for (co-)parallel and sequential independence and we have shown that provided that these conditions are satisfied, firing and transformation steps can be performed in any order, yielding the same result. Moreover, we have correlated these conditions, i.e. that parallel independence implies sequential independence and vice versa, sequential (coparallel) independence implies parallel and coparallel (parallel and sequential) independence. The advantage of the presented conditions is that they can be checked syntactically and locally instead of semantically and globally. Thus, they are also applicable in the case of complex reconfigurable P/T-systems.

Transformations of nets can be considered in various ways. Transformations of Petri nets to another Petri net class (e.g. in [7,10,35]), to another modeling technique or vice versa (e.g in [2,5,15,26,33,14]) are well examined and have yielded many important results. Transformation of one net into another without changing the net class is often used for purposes of forming a hierarchy, in terms of reductions or abstraction (e.g. in [22,16,20,12,8]) or transformations are used to detect specific properties of nets (e.g. in [3,4,6,29]). Net transformations that aim directly at changing the net in arbitrary ways as known from graph transformations were developed as a special case of high-level replacement systems e.g. in [17]. The general approach can be restricted to transformations that preserve specific properties as safety or liveness (see [30,32]). Closely related are those approaches that propose changing nets in specific ways in order to preserve specific semantic properties, as equivalent (I/O-) behavior (e.g in [1,11]), invariants (e.g. in [13]) or liveness (e.g. in [19,37]). Related are also those approaches that follow the "nets as tokens"-paradigm, based on elementary object nets introduced in [36]. Mobile object net systems [24,21] are an algebraic formalization of the elementary object nets that are closely related to our approach. In both cases the data types, respectively the colors represent the nets that are the token nets. Our approach goes beyond those approaches as we additionally have rules as tokens, and transformations of nets as operations. In [24] concurrency aspects between token nets have been investigated, but naturally not concerning net transformations. In [27] rewriting of Petri nets in terms of graph grammars are used for the reconfiguration of nets as well, but this approach lacks the ”nets as tokens”-paradigm.

In this paper we present main results of a line of research[2] concerning formal modeling and analysis of workflows in mobile ad-hoc networks. So, there is a large amount of most interesting and relevant open questions directly related to the work presented here. While a firing step and a transformation step that are parallel independent can be applied in any order, an aspect of future work is under which conditions they can be applied in parallel leading to the notions of parallel steps. Vice versa a parallel step should be splitted into the corresponding firing and transformation steps. This problem is closely related to the Parallelism Theorem for high-level replacement systems [17] which is the basis of a shift construction for transformation sequences. Moreover, it is most interesting to transfer further results which are already valid for high-level replacement systems, e.g. confluence, termination and critical pairs [17]. We plan to develop a tool for our approach using the graph transformation engine AGG[3] as a tool for the analysis of transformation properties like independence and termination, meanwhile the token net properties could be analyzed using the Petri Net Kernel [25], a tool infrastructure for Petri nets different net classes.

References

1. Balbo, G., Bruell, S., Sereno, M.: Product Form Solution for Generalized Stochastic Petri Nets. IEEE Transactions on Software Engineering 28(10), 915–932 (2002)
2. Belli, F., Dreyer, J.: Systems Modelling and Simulation by Means of Predicate/Transition Nets and Logic Programming. In: Proc. Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE), pp. 465–474 (1994)
3. Berthelot, G.: Checking properties of nets using transformation. In: Proc. Applications and Theory in Petri Nets. LNCS, vol. 222, pp. 19–40. Springer, Heidelberg (1985)
4. Berthelot, G.: Transformations and Decompositions of Nets. In: Petri Nets: Central Models and Their Properties, Part I, Advances in Petri Nets. LNCS, vol. 254, pp. 359–376. Springer, Heidelberg (1987)
5. Bessey, T., Becker, M.: Comparison of the modeling power of fluid stochastic Petri nets (FSPN) and hybrid Petri nets (HPN). In: Proc.Systems, Man and Cybernetics (SMC), vol. 2, pp. 354–358. IEEE Computer Society Press, Los Alamitos (2002)
6. Best, E., Thielke, T.: Orthogonal Transformations for Coloured Petri Nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 447–466. Springer, Heidelberg (1997)
7. Billington, J.: Extensions to Coloured Petri Nets. In: Proc. Petri Nets and Performance Models (PNPM), pp. 61–70. IEEE Computer Society Press, Los Alamitos (1989)

[2] The research project *Formal Modeling and Analysis of Flexible Processes in Mobile Ad-hoc Networks* (forMA$_1$NET) of the German Research Council.

[3] tfs.cs.tu-berlin.de/agg

8. Bonhomme, P., Aygalinc, P., Berthelot, G., Calvez, S.: Hierarchical control of time Petri nets by means of transformations. In: Proc. Systems, Man and Cybernetics (SMC), vol. 4, p. 6. IEEE Computer Society Press, Los Alamitos (2002)
9. Bottoni, P., De Rosa, F., Hoffmann, K., Mecella, M.: Applying Algebraic Approaches for Modeling Workflows and their Transformations in Mobile Networks. Journal of Mobile Information Systems 2(1), 51–76 (2006)
10. Campos, J., Sánchez, B., Silva, M.: Throughput Lower Bounds for Markovian Petri Nets: Transformation Techniques. In: Proc. Petri Nets and Performance Models (PNPM), pp. 322–331. IEEE Computer Society Press, Los Alamitos (1991)
11. Carmona, J., Cortadella, J.: Input/Output Compatibility of Reactive Systems. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 360–377. Springer, Heidelberg (2002)
12. Chehaibar, G.: Replacement of Open Interface Subnets and Stable State Transformation Equivalence. In: Proc. Applications and Theory of Petri Nets (ATPN). LNCS, vol. 674, pp. 1–25. Springer, Heidelberg (1991)
13. Cheung, T., Lu, Y.: Five Classes of Invariant-Preserving Transformations on Colored Petri Nets. In: Donatelli, S., Kleijn, J.H.C.M. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 384–403. Springer, Heidelberg (1999)
14. Cortés, L., Eles, P., Peng, Z.: Modeling and formal verification of embedded systems based on a Petri net representation. Journal of Systems Architecture 49(12-15), 571–598 (2003)
15. de Lara, J., Vangheluwe, H.: Computer Aided Multi-Paradigm Modelling to Process Petri-Nets and Statecharts. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 239–253. Springer, Heidelberg (2002)
16. Desel, J.: On Abstraction of Nets. In: Proc. Applications and Theory of Petri Nets (ATPN). LNCS, vol. 524, pp. 78–92. Springer, Heidelberg (1990)
17. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. In: EATCS Monographs in Theoretical Computer Science, Springer, Heidelberg (2006)
18. Ehrig, H., Hoffmann, K., Prange, U., Padberg, J.: Formal Foundation for the Reconfiguaration of Nets. Technical report, TU Berlin, Fak. IV (2007)
19. Esparza, J.: Model Checking Using Net Unfoldings. Science of Computer Programming 23(2-3), 151–195 (1994)
20. Esparza, J., Silva, M.: On the analysis and synthesis of free choice systems. In: Proc. Applications and Theory of Petri Nets (ATPN). LNCS, vol. 483, pp. 243–286. Springer, Heidelberg (1989)
21. Farwer, B., Köhler, M.: Mobile Object-Net Systems and their Processes. Fundamenta Informaticae 60(1–4), 113–129 (2004)
22. Haddad, S.: A Reduction Theory for Coloured Nets. In *Proc. Applications and Theory in Petri Nets (ATPN).* In: Proc. Applications and Theory in Petri Nets (ATPN). LNCS, vol. 424, pp. 209–235. Springer, Heidelberg (1988)
23. Hoffmann, K., Ehrig, H., Mossakowski, T.: High-Level Nets with Nets and Rules as Tokens. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 268–288. Springer, Heidelberg (2005)
24. Köhler, M., Rölke, H.: Concurrency for mobile object net systems. Fundamenta Informaticae 54(2-3), 221–235 (2003)
25. Kindler, E., Weber, M.: The Petri Net Kernel - An Infrastructure for Building Petri Net Tools. Software Tools for Technology Transfer 3(4), 486–497 (2001)

26. Kluge, O.: Modelling a Railway Crossing with Message Sequence Charts and Petri Nets. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 197–218. Springer, Heidelberg (2003)
27. Llorens, M., Oliver, J.: Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets. IEEE Transactions on Computers 53(9), 1147–1158 (2004)
28. Meseguer, J., Montanari, U.: Petri Nets Are Monoids. Information and Computation 88(2), 105–155 (1990)
29. Murata, T.: Petri nets: Properties, analysis and applications. In: Proc. IEEE, vol. 77, pp. 541 – 580. IEEE (1989)
30. Padberg, J., Gajewsky, M., Ermel, C.: Rule-based refinement of high-level nets preserving safety properties. Science of Computer Programming 40(1), 97–118 (2001)
31. Padberg, J., Hoffmann, K., Ehrig, H., Modica, T., Biermann, E., Ermel, C.: Maintaining Consistency in Layered Architectures of Mobile Ad-hoc Networks. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007, LNCS, vol. 4422, pp. 383–397, Springer, Heidelberg (2007)
32. Padberg, J., Urbášek, M.: Rule-Based Refinement of Petri Nets: A Survey. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 161–196. Springer, Heidelberg (2003)
33. Parisi-Presicce, F.: A Formal Framework for Petri Net Class Transformations. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 409–430. Springer, Heidelberg (2003)
34. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific, Singapore (1997)
35. Urbášek, M.: Categorical Net Transformations for Petri Net Technology. PhD thesis, Technische Universität Berlin (2003)
36. Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–25. Springer, Heidelberg (1998)
37. 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)

From Many Places to Few: Automatic Abstraction Refinement for Petri Nets[⋆]

Pierre Ganty[⋆⋆], Jean-François Raskin, and Laurent Van Begin[⋆⋆⋆]

Département d'Informatique, Université Libre de Bruxelles (U.L.B.)
{pganty,jraskin,lvbegin}@ulb.ac.be

Abstract. Current algorithms for the automatic verification of Petri nets suffer from the explosion caused by the high dimensionality of the state spaces of practical examples. In this paper, we develop an abstract interpretation based analysis that reduces the dimensionality of state spaces that are explored during verification. In our approach, the dimensionality is reduced by trying to gather places that may not be important for the property to establish. If the abstraction that is obtained is too coarse, an automatic refinement is performed and a more precise abstraction is obtained. The refinement is computed by taking into account information about the inconclusive analysis. The process is iterated until the property is proved to be true or false.

1 Introduction

Petri nets (and their monotonic extensions) are well-adapted tools for modeling concurrent and infinite state systems like, for instance, parameterized systems [1]. Even though their state space is infinite, several interesting problems are decidable on Petri nets. The seminal work of Karp and Miller [2] shows that, for Petri nets, an effective representation of the downward closure of the set of reachable markings, the so-called *coverability set*, is constructible. This coverability set is the main tool needed to decide several interesting problems and in particular the *coverability problem*. The coverability problem asks: "given a Petri net N, an initial marking m_0 and a marking m, is there a marking m' reachable from m_0 which is greater or equal to m". The coverability problem was shown decidable in the nineties for the larger class of *well-structured transition systems* [3, 4]. That class of transition systems includes a large number of interesting infinite state models including Petri nets and their monotonic extensions.

A large number of works have been devoted to the study of efficient techniques for the automatic verification of coverability properties of infinite state Petri nets, see for example [5, 6, 7, 8]. Forward and backward algorithms are now available and have been implemented to show their practical relevance. All those methods manipulate, somehow or other, infinite sets of markings. Sets of markings are subsets of $\mathbb{N}^k$ where

[⋆] This research was supported the Belgian FNRS grant 2.4530.02 of the FRFC project "Centre Fédéré en Vérification" and by the project "MoVES", an Interuniversity Attraction Poles Programme of the Belgian Federal Government.

[⋆⋆] Pierre Ganty is supported by the FNRS under a FRIA grant.

[⋆⋆⋆] Laurent Van Begin is "Chargé de recherche" at FNRS, Belgium.

$\mathbb{N}$ is the set of positive integers and k is the number of places in the Petri net. We call k its *dimension*. When k becomes large the above mentioned methods suffer from the *dimensionality problem*: the sets that have to be handled have large representations that make them hard to manipulate efficiently.

In this paper, we develop an automatic abstraction technique that attacks the dimensionality problem. To illustrate our method, let us consider the Petri net of Fig. 1(a). This Petri net describes abstractly a system that spawns an arbitrary number of processes running in parallel. There are two independent critical sections in the system that correspond to places p_4, p_5 and to places p_8, p_9. One may be interested in proving that mutual exclusion is ensured between p_4 and p_5. That mutual exclusion property is local to a small part of the net, and it is intuitively clear that the places $p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}$ are irrelevant to prove mutual exclusion between p_4 and p_5. Hence, the property can be proved with an abstraction of the Petri net as shown in Fig. 1(b) where the places $\{p_1, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}\}$ are not distinguished and merged into a single place p_1'. However, the current methods for solving coverability, when given the Petri net of Fig. 1(a) will consider the entire net and manipulate subsets of $\mathbb{N}^{12}$. Our method will automatically consider sets of lower dimensionality: in this case subsets of $\mathbb{N}^5$.

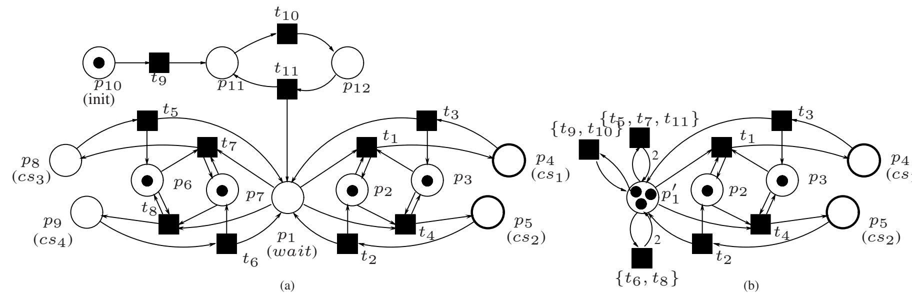


Fig. 1. A Petri net with two distinct mutual exclusion properties (a) and its abstraction (b)

Our algorithm is based on two main ingredients: abstract interpretation and automatic refinement. Abstract interpretation [11] is a well-established technique to define, in a systematic way, abstractions of semantics. In our case, we will use the notion of Galois insertion to relate formally subsets in $\mathbb{N}^k$ with their abstract representation in $\mathbb{N}^{k'}$ with $k' < k$. This Galois insertion allows us to systematically design an abstract semantics that leads to efficient semi-algorithms to solve the coverability problem by manipulating *lower dimensional sets*. We will actually show that the original coverability problem reduces to a coverability problem of lower dimensionality and so our algorithm can reuse efficient implementations for the forward and backward analysis of those abstractions. When the abstract interpretation is inconclusive, because it is not precise enough, our algorithm automatically refines the abstract domain. This refinement ensures that the next analysis will be more precise and that the abstract analysis will eventually be precise enough to decide the problem. The abstraction technique that we consider here uses all the information that has been computed by previous steps and is quite different from the technique known as *counterexample guided abstraction refinement* [9].

We have implemented our automatic abstraction technique and we have evaluated our new algorithm on several interesting examples of infinite state Petri nets taken from the literature. It turns out that our technique finds low dimensional systems that are sufficiently precise abstractions to establish the correctness of complex systems. We also have run our algorithm on finite state models of well-known mutual exclusion protocols. On those, the reduction in dimension is less spectacular but our algorithm still finds simplifications that would be very hard to find by hand.

To the best of our knowledge, this work is the first that tries to automatically abstract Petri nets by lowering their dimensionality and which provide an automatic refinement when the analysis is not conclusive. In [10], the authors provide syntactical criterion to simplify Petri nets while our technique is based on semantics. Our technique provides automatically much coarser abstractions than the one we could obtain by applying rules in [10].

Due to the lack of space the omitted proofs can be found in a technical report available at `http://www.ulb.ac.be/di/ssd/cfv/publications.html`.

2 Preliminaries and Outline

We start this section by recalling Petri nets, their semantics and the coverability problem. Then, we recall the main properties of existing algorithms to solve this coverability problem. We end the section by giving an outline of our new algorithm.

Petri Nets and Their (Concrete) Semantics. In the rest of the paper our model of computation is given by the Petri net formalism. Given a set S we denote by $|S|$ its cardinality.

Definition 1 (Petri nets). *A Petri net N is given by a tuple (P, T, F, m_0) where:*

- *P and T are finite disjoint sets of* places *and* transitions *respectively,*
- *$F = (\mathcal{I}, \mathcal{O})$ are two mappings: $\mathcal{I}, \mathcal{O} \colon P \times T \mapsto \mathbb{N}$ describing the relationship between places and transitions. Once a linear order has been fixed on P and on T, $\mathcal{I}$ and $\mathcal{O}$ can be seen as $(|P|, |T|)$-matrices over $\mathbb{N}$ ($\mathbb{N}^{|P|\times|T|}$ for short). Let $t \in T$, $\mathcal{I}(t)$ denote the t-column vector in $\mathbb{N}^{|P|}$ of $\mathcal{I}$.*
- *m_0 is the* initial marking. *A* marking *$m \in \mathbb{N}^{|P|}$ is a column vector giving a number $m(p)$ of tokens for each place $p \in P$.*

Throughout the paper we will use the letter k to denote $|P|$, i.e. the *dimensionality* of the net. We introduce the partial order $\preccurlyeq \subseteq \mathbb{N}^k \times \mathbb{N}^k$ such that for all $m, m' \in \mathbb{N}^k$: $m \preccurlyeq m'$ iff $m(i) \leq m'(i)$ for all $i \in [1..k]$ (where $[1..k]$ denotes the set $\{1, \ldots, k\}$). It turns out that $\preccurlyeq$ is a well-quasi order (wqo for short) on $\mathbb{N}^k$ meaning that for any infinite sequence of markings $m_1, m_2, \ldots, m_i, \ldots$ there exists indices $i < j$ such that $m_i \preccurlyeq m_j$.

Definition 2 (Firing Rules of Petri net). *Given a Petri net $N = (P, T, F, m_0)$ and a marking $m \in \mathbb{N}^k$ we say that the transition t is* enabled *at m, written $m(t\rangle$, iff $\mathcal{I}(t) \leqslant m$. If t is enabled at m then the* firing *of t at m leads to a marking m', written $m(t\rangle m'$, such that $m' = m - \mathcal{I}(t) + \mathcal{O}(t)$.*

Given a Petri net we are interested in the set of markings it can reach. To formalize the set of reachable markings and variants of the reachability problem, we use the following lattice and the following operations on sets of markings.

Definition 3. *Let $k \in \mathbb{N}$, the powerset lattice associated to $\mathbb{N}^k$ is the complete lattice $(\wp(\mathbb{N}^k), \subseteq, \cup, \cap, \emptyset, \mathbb{N}^k)$ having the powerset of $\mathbb{N}^k$ as a carrier, union and intersection as least upper bound and greatest lower bound operations, respectively and the empty set and $\mathbb{N}^k$ as the $\subseteq$-minimal and $\subseteq$-maximal elements, respectively.*

We use Church's lambda notation (so that F is $\lambda X.\, F(X)$) and use the composition operator $\circ$ on functions given by $(f \circ g)(x) = f(g(x))$. Also we define $f^{i+1} = f^i \circ f$ and $f^0 = \lambda x.\, x$. Sometimes we also use logical formulas. Given a logical formula ψ we write $[\![\psi]\!]$ for the set of its satisfying valuations.

Definition 4 (The predicate transformers pre, $\widetilde{pre}$, and $post$). *Let N a Petri net given by (P, T, F, m_0) and let $t \in T$, we define $pre_N[t], \widetilde{pre}_N[t], post_N[t] \colon \wp(\mathbb{N}^k) \mapsto \wp(\mathbb{N}^k)$ as follows,*

$$\begin{aligned} pre_N[t] &\overset{\text{def}}{=} \lambda X.\, \{m \in \mathbb{N}^k \mid \exists m' \colon m' \in X \wedge m(t\rangle m'\} \\ \widetilde{pre}_N[t] &\overset{\text{def}}{=} \lambda X.\, \{m \in \mathbb{N}^k \mid \mathcal{I}(t) \nleqslant m \vee m \in pre_N[t](X)\} \\ post_N[t] &\overset{\text{def}}{=} \lambda X.\, \{m' \in \mathbb{N}^k \mid \exists m \colon m \in X \wedge m(t\rangle m'\} \ . \end{aligned}$$

The extension to the set T of transitions is given by,

$$f_N = \begin{cases} \lambda X.\, \bigcup_{t \in T} f_N[t](X) & \textit{if } f_N \textit{ is } pre_N \textit{ or } post_N \\ \lambda X.\, \bigcap_{t \in T} f_N[t](X) & \textit{for } f_N = \widetilde{pre}_N.s \end{cases}$$

In the sequel when the Petri net N is clear from the context we omit to mention N as a subscript. Finally we recall a well-known result which is proved for instance in [12]: for any $X, Y \subseteq \mathbb{N}^k$ we have

$$post(X) \subseteq Y \Leftrightarrow X \subseteq \widetilde{pre}(Y) \ . \tag{Gc}$$

All those predicate transformers are monotone functions over the complete lattice $(\wp(\mathbb{N}^k), \subseteq, \cup, \cap, \emptyset, \mathbb{N}^k)$ so they can be used as building blocks to define fixpoints expressions.

In $\wp(\mathbb{N}^k)$, *upward-closed* and *downward-closed* sets are particularly interesting and are defined as follows. We define the operator $\downarrow$ (resp. $\uparrow$) as $\lambda X.\, \{x' \in \mathbb{N}^k \mid \exists x \colon x \in X \wedge x' \leqslant x\}$ (resp. $\lambda X.\, \{x' \in \mathbb{N}^k \mid \exists x \colon x \in X \wedge x \leqslant x'\}$). A set S is $\leqslant$-downward closed ($\leqslant$-dc-set for short), respectively $\leqslant$-upward closed ($\leqslant$-uc-set for short), iff $\downarrow S = S$, respectively $\uparrow S = S$. We define $DCS(\mathbb{N}^k)$ ($UCS(\mathbb{N}^k)$) to be the set of all $\leqslant$-dc-sets ($\leqslant$-uc-sets). Those closed sets have natural effective representations that are based on the fact that $\leqslant$ is a wqo. A set $M \subseteq \mathbb{N}^k$ is said to be *canonical* if for any distinct $x, y \in \mathbb{N}^k$ we have $x \not\leqslant y$. We say that M is a *minor set* of $S \subseteq \mathbb{N}^k$, if $M \subseteq S$ and $\forall s \in S\, \exists m \in M : m \leqslant s$.

Lemma 1 ([13]). *Given $S \subseteq \mathbb{N}^k$, S has exactly one finite canonical minor set.*

So, any $\leqslant$-uc-set can be represented by its finite set of minimal elements. Since any $\leqslant$-dc-set is the complement of a $\leqslant$-upward-closed, it has an effective representation. Sets of omega-markings is an equivalent alternative representation [2].

We now formally state the coverability problem for Petri nets which corresponds to a fixpoint checking problem. Our formulation follows [12].

Problem 1. Given a Petri net N and a $\leqslant$-dc-set S, we want to check if the inclusion holds:

$$\mathit{lfp}\lambda X.\, \{m_0\} \cup \mathit{post}(X) \subseteq S \tag{1}$$

which, by [12, Thm. 4], is equivalent to

$$\{m_0\} \subseteq \mathit{gfp}\lambda X.\, S \cap \widetilde{\mathit{pre}}(X)\ \ . \tag{2}$$

We write $\mathit{post}^*(m_0)$ and $\widetilde{\mathit{pre}}^*(S)$ to be the fixpoints of relations (1) and (2), respectively. They are called the *forward semantics* and the *backward semantics* of the net. Note also that since S is a $\leqslant$-dc-set, $\mathit{post}^*(m_0) \subseteq S$ if and only if $\big\downarrow(\mathit{post}^*(m_0)) \subseteq S$.

Existing Algorithms. The solutions to problem 1 found in the literature (see [14, 15]) iteratively compute finer overapproximations of $\big\downarrow(\mathit{post}^*(m_0))$. They end up with an overapproximation $\mathcal{R}$ satisfying the following properties:

$$\mathit{post}_N^*(m_0) \subseteq \mathcal{R} \tag{A1}$$
$$\mathcal{R} \in DCS(\mathbb{N}^k) \tag{A2}$$
$$\mathit{post}_N(\mathcal{R}) \subseteq \mathcal{R} \tag{A3}$$
$$\mathit{post}_N^*(m_0) \subseteq S \rightarrow \mathcal{R} \subseteq S \tag{A4}$$

The solutions of [14, 15] actually solve problem 1 for the entire class of well-structured transition systems (WSTS for short) which includes Petri nets and many other interesting infinite state models. In [2] the authors show that $\big\downarrow(\mathit{post}^*(m_0))$ is computable for Petri nets and thus the approximation scheme presented above also encompasses this solution. All these solutions have in input an effective representation for (1) the initial marking m_0, (2) the predicate transformer post_N associated to the Petri net N and (3) the $\leqslant$-dc-set S.

In the literature, see for example [4], there are also solutions which compute the set $\widetilde{\mathit{pre}}^*(S)$ by evaluating its associated fixpoint (see (2)). Since [4], this fixpoint is known to be computable for WSTS and thus also for Petri net.[1]

[1] The fixpoint expression considered in [4] is actually different from (2) but coincides with its complement.

All these algorithms for Petri nets suffer from the explosion caused by the high dimensionality of the state spaces of practical examples. In this paper, we develop an analysis that reduces the dimensionality of state spaces that are explored during verification.

Overview of Our Approach. In order to mitigate the dimensionality problem, we adopt the following strategy. First, we define a parametric abstract domain where subsets of $\mathbb{N}^k$ are abstracted by subsets of $\mathbb{N}^{k'}$ where $k' < k$ (k' being a parameter). More precisely, when each dimension in the concrete domain records the number of tokens contained in each place of the Petri net, in the abstract domain, each dimension records the sum of the number of tokens contained into a set of places. Using this abstract domain, we define abstract forward and abstract backward semantics, and define efficient algorithms to compute them. In those semantics, sets of markings are represented by subsets of $\mathbb{N}^{k'}$. If the abstract semantics is not conclusive, it is refined automatically using a refinement procedure that is guided by the inconclusive abstract semantics. During the refinement steps, we identify important concrete sets and refine the current abstract domain to allow the exact representation of those sets.

The rest of our paper formalizes those ideas and is organized as follows. In Sect. 3, we define our parametric abstract domain and we specify the abstract semantics. We also show how the precision of different domains of the family can be related. In Sect. 4, we define an efficient way to overapproximate the abstract semantics defined in Sect. 3. In Section 5 shows how to refine automatically abstract domains. We define there an algorithm that given a concrete set M computes the coarsest abstract domain that is able to represent M exactly. In Sect. 6, we put all those results together to obtain our algorithm that decides coverability by successive approximations and refinements. In Sect. 7, we report on experiments that show the practical interest of our new algorithm.

3 Abstraction of Sets of Markings

Partitions. At the basis of our abstraction technique are the *partitions* (of the set of places).

Definition 5. *Let A be a partition of the set $[1..k]$ into k'* classes *$\{C_i\}_{i\in[1..k']}$. We define the order $\preceq$ over partitions as follows: $A \preceq A'$ iff $\forall C \in A\, \exists C' \in A' \colon C \subseteq C'$. It is well known, see [16], that the set of partitions of $[1..k]$ together with $\preceq$ form a complete lattice where $\{\{1\}, \ldots, \{k\}\}$ is the $\preceq$-minimal partition, $\{\{1, \ldots, k\}\}$ is the $\preceq$-maximal partition and the greatest lower bound of two partitions A_1 and A_2, noted $A_1 \curlywedge A_2$ is the partition given by $\{C \mid \exists C_1 \in A_1\, \exists C_2 \in A_2 \colon C = C_1 \cap C_2 \wedge C \neq \emptyset\}$. The least upper bound of two partitions A_1 and A_2, noted $A_1 \curlyvee A_2$ is the finest partition such that given $C \in A_1 \cup A_2$ and $\{a_1, a_2\} \subseteq C$ we have $\exists C' \in A_1 \curlyvee A_2 \colon \{a_1, a_2\} \subseteq C'$.*

Partitions will be used to abstract sets of markings by lowering their dimensionality. Given a marking m (viz. a k-uple) and a partition A of $[1..k]$ into k' classes we abstract m into a k'-uple m' by taking the sum of all the coordinates of each class. A simple way to apply the abstraction on a marking m is done by computing the product of a matrix A with the vector of m (noted $A{\cdot}\, m$). So we introduce a matrix based definition for partitions.

Definition 6. *Let A be a partition of $[1..k]$ given by $\{C_i\}_{i\in[1..k']}$. We associate to this partition a matrix $A := (a_{ij})_{k'\times k}$ such that $a_{ij} = 1$ if $j \in C_i$, $a_{ij} = 0$ otherwise. So, $A \in \{0,1\}^{k'\times k}$. We write $\mathcal{A}^{k'\times k}$ to denote the set of matrices associated to the partitions of $[1..k]$ into k' classes.*

We sometimes call such a A an *abstraction*.

Abstract Semantics. We are now equipped to define an abstraction technique for sets of markings. Then we focus on the abstraction of the predicate transformers involved in the fixpoints of (1) and (2).

Definition 7. *Let $A \in \mathcal{A}^{k'\times k}$, we define the abstraction function $\alpha_A\colon \wp(\mathbb{N}^k) \mapsto \wp(\mathbb{N}^{k'})$ and the concretization function $\gamma_A\colon \wp(\mathbb{N}^{k'}) \mapsto \wp(\mathbb{N}^k)$ respectively as follows*

$$\alpha_A \stackrel{\text{def}}{=} \lambda X.\, \{A{\cdot}\, x \mid x \in X\} \qquad \gamma_A \stackrel{\text{def}}{=} \lambda X.\, \{x \mid A{\cdot}\, x \in X\}\ .$$

In the following, if A is clear from the context, we will write α (resp. γ) instead of α_A (resp. γ_A). Given the posets $\langle L, \leqslant\rangle$ and $\langle M, \sqsubseteq\rangle$ and the maps $\alpha \in L \mapsto M$, $\gamma \in M \mapsto L$, we write $\langle L, \leqslant\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \sqsubseteq\rangle$ if they form a Galois insertion [11], that is $\forall x \in L, \forall y \in M\colon \alpha(x) \sqsubseteq y \Leftrightarrow x \leqslant \gamma(y)$ and $\alpha \circ \gamma = \lambda x.\, x$.

Proposition 1. *Let $A \in \mathcal{A}^{k'\times k}$, we have $(\wp(\mathbb{N}^k), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\wp(\mathbb{N}^{k'}), \subseteq)$.*

Proof. From Def. 7 it is clear that α and γ are monotone functions.
Let $X \subseteq \mathbb{N}^k$ and $Y \subseteq \mathbb{N}^{k'}$,

$$\begin{aligned}
&\alpha(X) \subseteq Y \\
&\Leftrightarrow \{A{\cdot}\, x \mid x \in X\} \subseteq Y && \text{def. 7} \\
&\Leftrightarrow \forall x\colon x \in X \rightarrow A{\cdot}\, x \in Y \\
&\Leftrightarrow X \subseteq \{x \mid A.x \in Y\} \\
&\Leftrightarrow X \subseteq \gamma(Y) && \text{def. 7}
\end{aligned}$$

We now prove that $\alpha \circ \gamma(Y) = Y$. First, note that for all $y \in Y$ there exists $x \in \gamma(y)$. In particular, given $y \in Y$ we define $x \in \mathbb{N}^k$ such that for any $i \in [1..k']$ all components of class C_i equals to 0 but one component which equals to $y(i)$. It is routine to check that $A \cdot x = y$, i.e. $x \in \gamma(y)$.

$$\begin{aligned}
\alpha \circ \gamma(Y) &= \alpha(\{x \mid A{\cdot}\, x \in Y\}) \\
&= \{A{\cdot}\, x \mid A{\cdot}\, x \in Y\} && \text{def. 7} \\
&= Y && \text{by above}
\end{aligned}$$

□

Given a Galois insertion, the theory of abstract interpretation [11] provides us with a theoretical framework to systematically derive approximate semantics. The concrete forward semantics of a Petri net N is given by $post^*_N(m_0)$. Since we have a Galois insertion, $post^*_N(m_0)$ has a unique best approximation in the abstract domain. This value is $\alpha(post^*_N(m_0))$.

Unfortunately, there is no general method to compute this approximation without computing $post^*_N(m_0)$ first. So instead of trying to compute this abstract value, we compute an overapproximation. Let $\mathcal{F}$ be an overapproximation of $\alpha(post^*_N(m_0))$ and let $\mathcal{B}$ be an overapproximation $\alpha(\widetilde{pre}^*_N(S))$. The following lemma shows the usefulness of such approximations.

Lemma 2. *Given a Petri net N and a $\leqslant$-dc-set S we have*

$$\gamma(\mathcal{F}) \subseteq S \rightarrow post^*_N(m_0) \subseteq S$$
$$\{m_0\} \nsubseteq \gamma(\mathcal{B}) \rightarrow post^*_N(m_0) \nsubseteq S$$

Abstract interpretation [11] tells us that to compute an overapproximation of fixpoints of a concrete function, we must first approximate this function by an abstract function and compute the fixpoint of this abstract function in the abstract domain. Among the abstractions of a function f is the most precise one. In [11] the authors show that, in the context of a Galois insertion, themost precise approximation of f is unique and given by $\alpha \circ f \circ \gamma$. So to approximate $\alpha(post^*_N(m_0))$ and $\alpha(\widetilde{pre}^*_N(S))$ we obtain the following fixpoint expression in the abstract domain:

$$lfp\lambda X.\,\alpha(\{m_0\} \cup post(\gamma(X))) \qquad \text{and} \qquad gfp\lambda X.\,\alpha(S \cap \widetilde{pre}(\gamma(X))) \ , \qquad (3)$$

respectively. This definition naturally suggests to concretize the argument, then apply f and finally to abstract its result. In practice applying this methodology leads to inefficient algorithms. Indeed the explicit computation of γ is in general costly. In our settings it happens that given an effective representation of M the effective representation of the set $\gamma(M)$ could be exponentially larger. In fact, let A be a partition of $[1..k]$ given by $\{C_i\}_{i\in[1..k']}$ and let $\hat{m} \in \mathbb{N}^{k'}$, we have $|\gamma(\hat{m})| = \prod_{i\in[1..k']} \binom{\hat{m}(i)+|C_i|-1}{|C_i|-1}$. Section 4 is devoted to the definition of the most precise approximation without explicitly evaluating γ.

Refinement. As mentioned in Sect. 2, our algorithm is based on the abstraction refinement paradigm. In that context, if the current abstraction A_i is inconclusive we refine it into an abstraction A_{i+1} which overapproximates sets of markings and predicate transformers more precisely than A_i. Here follows a result relating the precision of abstractions with their underlying partitions.

Lemma 3. *Let A, A' be two partitions of $[1..k]$ such that $A \preceq A'$ and $M \subseteq \mathbb{N}^k$,*

$$\gamma_A \circ \alpha_A(M) \subseteq \gamma_{A'} \circ \alpha_{A'}(M) \ .$$

So by refining partitions, we refine abstractions. We will see in Sect. 5 how to use this result systematically in our algorithm and how to take into account previous computations when a refinement is done. The following result tells us that if two partitions are able to represent exactly a set then their *lub* is also able to represent that set. So, for any set M there is a coarsest partition which is able to represent it.

Lemma 4. *Let* A, A_1, A_2 *be three partitions of* $[1..k]$ *such that* $A = A_1 \curlyvee A_2$ *and* $M \subseteq \mathbb{N}^k$*, we have*

$$\textit{if} \left\{ \begin{array}{l} \gamma_{A_1} \circ \alpha_{A_1}(M) \subseteq M \\ \gamma_{A_2} \circ \alpha_{A_2}(M) \subseteq M \end{array} \right\} \textit{then}\ \gamma_A \circ \alpha_A(M) \subseteq M \ . \tag{4}$$

Proof. First given an abstraction A, we define $\mu_A = \gamma_A \circ \alpha_A$. Let $m \in M$ and $m' \in \mu_A(\{m\})$. We will show that there exists a finite sequence $\mu_{A_{i_1}}, \mu_{A_{i_2}}, \ldots, \mu_{A_{i_n}}$ such that $m' \in \mu_{A_{i_1}} \circ \mu_{A_{i_2}} \circ \ldots \circ \mu_{A_{i_n}}(\{m\})$ and $\forall j \in [1..n] \colon i_j \in [1..2]$. Then we will conclude that $m' \in M$ by left hand side of (4).

It is well known that given a set S, the set of partitions of S coincides with the set of equivalence classes in S. So we denote by $\equiv_A$ the equivalence relation defined by the partition A.

We thus get $m' \in \mu_A(\{m\})$ iff m' is obtained from m by moving tokens inside the equivalence classes of $\equiv_A$. More precisely, let $v \in \mathbb{N}$, and a, b two distinct elements of $[1..k]$ such that $\langle a, b\rangle \in \equiv_A$ and two markings $m_1, m_2 \in \mathbb{N}^k$ such that

$$m_2(q) = \begin{cases} m_1(q) + v & \text{if } q = a \\ m_1(q) - v & \text{if } q = b \\ m_1(q) & \text{otherwise.} \end{cases}$$

Intuitively the marking m_2 is obtained from m_1 by moving v tokens from b into a. So, since on one hand b and a belong to the same equivalence class and, on the other hand m_2 and m_1 contain an equal number of tokens we find that $m_2 \in \mu_A(\{m_1\})$.

Now we use the result of [16, Thm. 4.6] over the equivalence classes of a set. The theorem states that $\langle a, b\rangle \in \equiv_A$ iff there is a sequence of elements $c_1, \ldots, c_{n'}$ of $[1..k]$ such that

$$\langle c_i, c_{i+1}\rangle \in \equiv_{A_1} \qquad \text{or} \qquad \langle c_i, c_{i+1}\rangle \in \equiv_{A_2} \tag{5}$$

for $i \in [1..n'-1]$ and $a = c_1$, $b = c_{n'}$. From $c_1, \ldots, c_{n'}$ we define a sequence of n' moves whose global effect is to move v tokens from b into a. So given m_1, the marking obtained by applying this sequence of n' moves is m_2. Moreover, by eq. (5) we have that each move of the sequence is defined inside an equivalence class of $\equiv_{A_1}$ or $\equiv_{A_2}$. Hence each move of the sequence can be done using operator μ_{A_1} or μ_{A_2}.

Repeated application of the above reasoning shows that m' is obtained by moving tokens of m where moves are given by operators μ_{A_1} and μ_{A_2}. Formally this finite sequence of moves $\mu_{A_{i_1}}, \mu_{A_{i_2}}, \ldots, \mu_{A_{i_n}}$ is such that

$$\forall j \in [1..n] \colon i_j \in [1..2] \quad \text{and} \quad m' \in \mu_{A_{i_1}} \circ \mu_{A_{i_2}} \circ \ldots \circ \mu_{A_{i_n}}(\{m\}) \ .$$

Finally, left hand side of (4) and monotonicity of μ_{A_1}, μ_{A_2} shows that $m' \in M$. □

4 Efficient Abstract Semantics

In this section, we show how to compute a precise overapproximation of the abstract semantics efficiently without evaluating the concretization function γ. For that, we show that to any Petri net N of dimensionality k and any abstraction $A \in \mathcal{A}^{k' \times k}$, we can associate a Petri net $\hat{N}$ of dimensionality k' whose concrete forward and backward semantics gives precise overapproximations of the abstraction by A of the semantics of N given by (3).

Abstract Net. In order to efficiently evaluate the best approximation of $post_N[t]$ and $\widetilde{pre}_N[t]$ for each $t \in T$, without explicitly evaluating γ, we associate for each N and A a Petri net $\hat{N}$.

Definition 8. *Let N be a Petri net given by (P, T, F, m_0) and let $A \in \mathcal{A}^{k' \times k}$. We define the tuple $(\hat{P}, T, \hat{F}, \widehat{m_0})$ where*

- *$\hat{P}$ is a set of k' places (one for each class of the partition A),*
- *$\hat{F} = (\widehat{\mathcal{I}}, \widehat{\mathcal{O}})$ is such that $\widehat{\mathcal{I}} \overset{\text{def}}{=} A \cdot \mathcal{I}$ and $\widehat{\mathcal{O}} \overset{\text{def}}{=} A \cdot \mathcal{O}$,*
- *$\widehat{m_0}$ is given by $A \cdot m_0$.*

The tuple is a Petri net since $\widehat{m_0} \in \mathbb{N}^{|\hat{P}|}$, and $\widehat{\mathcal{I}}, \widehat{\mathcal{O}} \in \mathbb{N}^{(|\hat{P}|,|T|)}$. We denote by $\hat{N}$ this Petri net.

To establish properties of the semantics of this abstract net (given in Prop. 2 and Prop. 3 below), we need the technical results:

Lemma 5. *Let $A \in \mathcal{A}^{k' \times k}$ and $X \subseteq \mathbb{N}^k$,*

$$\gamma \circ \big\downarrow(X) = \downarrow \circ\, \gamma(X) \qquad \textit{and} \qquad \alpha \circ \big\downarrow(X) = \downarrow \circ\, \alpha(X) \ .$$

In the sequel we use the property that the abstraction function α is *additive* (i.e. $\alpha(A \cup B) = \alpha(A) \cup \alpha(B)$) and that γ is *co-additive* (i.e. $\gamma(A \cap B) = \gamma(A) \cap \gamma(B)$).

Forward Overapproximation. The next proposition states that the most precise approximation of the predicate transformer $post_N$ is given by the predicate transformer $post_{\hat{N}}$ of the abstract net.

Proposition 2. *Given a Petri net $N = (P, T, F, m_0)$, $A \in \mathcal{A}^{k' \times k}$ and $\hat{N}$ the Petri net given by def. 8, we have*

$$\lambda X.\, \alpha \circ post_N \circ \gamma(X) = \lambda X.\, post_{\hat{N}}(X) \ .$$

Proof. Definition 4 states that $post_N = \lambda X. \bigcup_{t \in T} post_N[t](X)$. Thus, for $t \in T$, we show that $\alpha \circ post_N[t] \circ \gamma(\hat{m}) = post_{\hat{N}}[t](\hat{m})$. Then the additivity of α shows the desired result.

For each $t \in T$, for each $\hat{m} \in \mathbb{N}^{k'}$,

$$\begin{aligned}
&\alpha \circ post_N[t] \circ \gamma(\hat{m}) \\
&= \alpha \circ post_N[t](\{m \mid m \in \gamma(\hat{m})\}) \\
&= \alpha(\{m - \mathcal{I}(t) + \mathcal{O}(t) \mid m \in \gamma(\hat{m}) \wedge \mathcal{I}(t) \leqslant m\}) && \text{def. 2} \\
&= \{A\cdot (m - \mathcal{I}(t) + \mathcal{O}(t)) \mid m \in \gamma(\hat{m}) \wedge \mathcal{I}(t) \leqslant m\} && \text{def. 7} \\
&= \{A\cdot m - A\cdot\mathcal{I}(t) + A\cdot\mathcal{O}(t) \mid m \in \gamma(\hat{m}) \wedge \mathcal{I}(t) \leqslant m\} \\
&= \{\alpha(m) - A\cdot\mathcal{I}(t) + A\cdot\mathcal{O}(t) \mid m \in \gamma(\hat{m}) \wedge \mathcal{I}(t) \leqslant m\} && \text{def of } \alpha \\
&= \{\hat{m} - A\cdot\mathcal{I}(t) + A\cdot\mathcal{O}(t) \mid m \in \gamma(\hat{m}) \wedge \mathcal{I}(t) \leqslant m\} && \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \\
&= \{\hat{m} - \widehat{\mathcal{I}}(t) + \widehat{\mathcal{O}}(t) \mid m \in \gamma(\hat{m}) \wedge \mathcal{I}(t) \leqslant m\} && \text{def. 8} \\
&= \{\hat{m} - \widehat{\mathcal{I}}(t) + \widehat{\mathcal{O}}(t) \mid \{\mathcal{I}(t)\} \subseteq {\downarrow} \circ \gamma(\hat{m})\} && \text{def of } \downarrow \\
&= \{\hat{m} - \widehat{\mathcal{I}}(t) + \widehat{\mathcal{O}}(t) \mid \{\mathcal{I}(t)\} \subseteq \gamma \circ {\downarrow}(\hat{m})\} && \text{Lem. 5} \\
&= \{\hat{m} - \widehat{\mathcal{I}}(t) + \widehat{\mathcal{O}}(t) \mid \alpha(\{\mathcal{I}(t)\}) \subseteq {\downarrow}(\hat{m})\} && \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \\
&= \{\hat{m} - \widehat{\mathcal{I}}(t) + \widehat{\mathcal{O}}(t) \mid \widehat{\mathcal{I}}(t) \leqslant \hat{m}\} && \text{def. 8} \\
&= post_{\hat{N}}[t](\hat{m}) && \text{def. 2} \quad \square
\end{aligned}$$

The consequences of Prop 2 are twofold. First, it gives a way to compute $\alpha \circ post_N \circ \gamma$ without computing explicitly γ and second since $post_{\hat{N}} = \alpha \circ post_N \circ \gamma$ and $\widehat{N}$ is a Petri net we can use any state of the art tool to check whether $post^*_{\hat{N}}(\widehat{m_0}) \subseteq \alpha(S)$ and conclude, provided $\gamma \circ \alpha(S) = S$, that $\gamma(post^*_{\hat{N}}(\widehat{m_0})) \subseteq S$, hence that $post^*_N(m_0) \subseteq S$ by Lem. 2.

Backward Overapproximation. The backward semantics of each transition of the abstract net is the best approximation of the backward semantics of each transitions of the concrete net. However, the best abstraction of the predicate transformer $\widetilde{pre}_N$ does not coincide with $\widetilde{pre}_{\hat{N}}$ as we will see later. To obtain those results, we need some intermediary lemmas (i.e. Lem. 6, 7 and 8).

Lemma 6. *Given a Petri net $N = (P, T, F, m_0)$ and $A \in \mathcal{A}^{k' \times k}$ we have*

$$\lambda X.\, \alpha \circ pre_N \circ \gamma(X) = \lambda X.\, pre_{\hat{N}}(X) \ .$$

Proof. The proof is similar to the proof of Prop. 2 with $\mathcal{O}$ (resp. $\widehat{\mathcal{O}}$) replaced by $\mathcal{I}$ (resp. $\widehat{\mathcal{I}}$) and vice versa. $\square$

Lemma 7. *Given a Petri net $N = (P, T, F, m_0)$ and a partition $A = \{C_j\}_{j\in[1..k']}$ of $[1..k]$, if $\exists i \in [1..k] \colon \mathcal{I}(i, t) > 0$ and $\{i\} \notin A$ then $\alpha(\{m \in \mathbb{N}^k \mid \mathcal{I}(t) \nleqslant m\}) = \mathbb{N}^{k'}$.*

Proof. Besides the hypothesis assume $i \in C_j$ and consider $l \in [1..k]$ such that $l \in C_j$ and $l \neq i$. The set $\{m \in \mathbb{N}^k \mid \mathcal{I}(t) \nleqslant m\}$ is a $\leqslant$-dc-set given by the following formula:

$$\bigvee_{\substack{p\in[1..k] \\ \mathcal{I}(p,t)>0}} x_p < \mathcal{I}(p, t).$$

We conclude from $i \in [1..k]$ and $\mathcal{I}(i,t) > 0$ that $[\![x_i < \mathcal{I}(i,t)]\!] = \{\langle v_1, \ldots, v_i, \ldots, v_k\rangle \mid v_i < \mathcal{I}(i,t)\}$, hence that $\alpha([\![x_i < \mathcal{I}(i,t)]\!]) = \mathbb{N}^{k'}$ by $\{i,l\} \subseteq C_j \in A$, and finally that $\alpha([\![x_i < \mathcal{I}(i,t)]\!]) \subseteq \alpha([\![\bigvee_{\substack{p\in[1..k]\\ \mathcal{I}(p,t)>0}} x_p < \mathcal{I}(p,t)]\!])$ by additivity of α. It follows that

$$\alpha(\{m \in \mathbb{N}^k \mid \mathcal{I}(t) \nleqslant m\}) = \alpha([\![\bigvee_{\substack{p\in[1..k]\\ \mathcal{I}(p,t)>0}} x_p < \mathcal{I}(p,t)]\!]) = \mathbb{N}^{k'}. \qquad \square$$

Lemma 8. *Given a Petri net $N = (P,T,F,m_0)$, a partition $A = \{C_j\}_{j\in[1..k']}$ of $[1..k]$ and $\hat{N}$ the Petri net given by def. 8, if for any $i \in [1..k]$: $\mathcal{I}(i,t) > 0$ implies $\{i\} \in A$, then $\alpha(\{m \in \mathbb{N}^k \mid \mathcal{I}(t) \nleqslant m\}) = \{m \in \mathbb{N}^{k'} \mid \widehat{\mathcal{I}}(t) \nleqslant m)\}$.*

Proof

$$\begin{aligned}
&\alpha(\{m \in \mathbb{N}^k \mid \mathcal{I}(t) \nleqslant m\}) \\
&= \{A\cdot m \mid m \in \mathbb{N}^k \wedge \mathcal{I}(t) \nleqslant m\}) && \text{def. of } \alpha\\
&= \{A\cdot m \mid m \in \mathbb{N}^k \wedge A\cdot\mathcal{I}(t) \nleqslant A\cdot m\} && \text{hyp.}\\
&= \{A\cdot m \mid m \in \mathbb{N}^k \wedge \widehat{\mathcal{I}}(t) \nleqslant A\cdot m\} && \text{def. of } \widehat{\mathcal{I}}\\
&= \{\widehat{m} \in \mathbb{N}^{k'} \mid \exists m \in \mathbb{N}^k : \widehat{m} = A\cdot m \wedge \widehat{\mathcal{I}}(t) \nleqslant \widehat{m}\}\\
&= \{\widehat{m} \in \mathbb{N}^{k'} \mid \widehat{\mathcal{I}}(t) \nleqslant \widehat{m}\} && \text{tautology} \qquad \square
\end{aligned}$$

We are now ready to state and prove that $\widetilde{pre}_{\hat{N}}[t]$ is the best approximation of $\widetilde{pre}_N[t]$.

Proposition 3. *Given a Petri net $N = (P,T,F,m_0)$, a partition $A = \{C_j\}_{j\in[1..k']}$ of $[1..k]$ and $\hat{N}$ the Petri net given by def. 8, we have*

$$\lambda X.\, \alpha \circ \widetilde{pre}_N[t] \circ \gamma(X) = \begin{cases} \mathbb{N}^{k'} & \textit{if } \exists i \in [1..k'] \colon |C_i| > 1 \wedge \widehat{\mathcal{I}}(i,t) > 0\\ \lambda X.\, \widetilde{pre}_{\hat{N}}[t](X) & \textit{otherwise.}\end{cases}$$

Proof

$$\begin{aligned}
&\alpha \circ \widetilde{pre}_N[t] \circ \gamma(S)\\
&= \alpha \circ \widetilde{pre}_N[t](\{m \in \mathbb{N}^k \mid m \in \gamma(S)\})\\
&= \alpha(\{m \mid (\mathcal{I}(t) \nleqslant m) \vee (\mathcal{I}(t) \leqslant m \wedge m - \mathcal{I}(t) + \mathcal{O}(t) \in \gamma(S))\}) && \text{def. of } \widetilde{pre}_N[t]\\
&= \alpha(\{m \mid \mathcal{I}(t) \nleqslant m\}) \cup \alpha(\{m \mid \mathcal{I}(t) \leqslant m \wedge m - \mathcal{I}(t) + \mathcal{O}(t) \in \gamma(S))\}) && \text{additivity of } \alpha\\
&= \alpha(\{m \mid \mathcal{I}(t) \nleqslant m\}) \cup \alpha \circ pre_N[t] \circ \gamma(S) && \text{def of } pre_N[t]\\
&= \alpha(\{m \mid \mathcal{I}(t) \nleqslant m\}) \cup pre_{\hat{N}}[t](S) && \text{by Lem. 6}
\end{aligned}$$

We now consider two cases:

- $\exists i \in [1..k] \colon \mathcal{I}(i,t) > 0$ and $\{i\} \notin A$. From Lemma 7, we conclude that $\alpha \circ \widetilde{pre}_N[t] \circ \gamma(S) = \mathbb{N}^{k'}$;
- $\forall i \in [1..k] \colon \mathcal{I}(i,t) > 0$ implies $\{i\} \in A$. In this case we have

$$\begin{aligned}
\alpha \circ \widetilde{pre}_N[t] \circ \gamma(S) &= \{m \in \mathbb{N}^{k'} \mid \widehat{\mathcal{I}}(t) \nleqslant m\} \cup pre_{\hat{N}}[t](S) && \text{by Lem. 8}\\
&= \widetilde{pre}_{\hat{N}}[t](S) && \text{def of } \widetilde{pre}_{\hat{N}}[t]
\end{aligned}$$

$\square$

Now, let us see how to approximate $\widetilde{pre}_N$. We can do that by distributing α over $\cap$ as shown below at the cost of an overapproximation:

$$\begin{aligned}
& gfp(\lambda X.\, \alpha(S \cap \widetilde{pre}_N \circ \gamma(X))) && (3)\\
&= gfp(\lambda X.\, \alpha(S \cap \bigcap_{t \in T} \widetilde{pre}_N[t] \circ \gamma(X))) && \text{def of } \widetilde{pre}_N\\
&\subseteq gfp(\lambda X.\, \alpha(S) \cap \bigcap_{t \in T} \alpha \circ \widetilde{pre}_N[t] \circ \gamma(X)) && \alpha(A \cap B) \subseteq \alpha(A) \cap \alpha(B)
\end{aligned}$$

Thus, this weaker result for the backward semantics stems from the definition of $\widetilde{pre}_N$ given by $\bigcap_{t \in T} \widetilde{pre}_N[t]$ and the fact that α is not *co-additive* (i.e. $\alpha(A \cap B) \neq \alpha(A) \cap \alpha(B)$).

5 Abstraction Refinement

In the abstraction refinement paradigm, if the current abstraction A_i is inconclusive it is refined. A refinement step will then produce an abstraction A_{i+1} which overapproximates sets of markings and predicate transformers more precisely than A_i.

We showed in Lem. 3 that if partition A refines A' (i.e. $A \prec A'$) then A represents sets of markings (and hence function over sets of markings) more precisely than A'. Note also that the partition A where each class is a singleton (i.e. the $\preceq$-minimal partition) we have $\gamma_A \circ \alpha_A(S) = S$ for any set of markings S. Thus the loss of precision stems from the classes of the partition which are not singleton.

With these intuitions in mind we will refine an abstraction A_i into A_{i+1} by splitting classes of A_i which are not singleton. A first idea to refine abstraction A_i is to split a randomly chosen non singleton class of A_i. This approach is complete since it will eventually end up with the $\preceq$-minimal partition which yields to a conclusive analysis with certainty. However, we adopt a different strategy which consists in computing for A_{i+1} the coarsest partition refining A_i and which is able to represent precisely a given set of markings.

Now we present the algorithm refinement that given a set of markings M computes the coarsest partition A which is able to represent M precisely. The algorithm starts from the $\preceq$-minimal partition then the algorithm chooses non-deterministically two candidate classes and merge them in a unique class. If this new partition still represents M precisely, we iterate the procedure. Otherwise the algorithm tries choosing different candidates. The algorithm is presented in Alg. 1.

Let $A = \{C_i\}_{i \in [1..k']}$ be a partition of $[1..k]$, we define $A_{C_i} = \{C_i\} \cup \{\{s\} \mid s \in [1..k] \wedge s \notin C_i\}$. We first prove the following lemma.

Lemma 9. *Let $A = \{C_i\}_{i \in [1..k']}$ be a partition of $[1..k]$, $M \subseteq \mathbb{N}^k$, we have:*

$$\gamma_A \circ \alpha_A(M) \subseteq M \Leftrightarrow \bigwedge_{C_i \in A} \gamma_{A_{C_i}} \circ \alpha_{A_{C_i}}(M) \subseteq M \ .$$

Algorithm 1. refinement

```
Input: $M \subseteq \mathbb{N}^k$
Output: a partition $A$ of $[1..k]$ such that $\gamma_A \circ \alpha_A(M) \subseteq M$
Let $A$ be $\{\{1\},\{2\},\dots,\{k\}\}$ ;
while $\exists C_i, C_j \in A : C_i \neq C_j$ and $\gamma_{A_{C_i \cup C_j}} \circ \alpha_{A_{C_i \cup C_j}}(M) \subseteq M$ do
1     Let $C_i, C_j \in A$ such that $C_i \neq C_j$ and $\gamma_{A_{C_i \cup C_j}} \circ \alpha_{A_{C_i \cup C_j}}(M) \subseteq M$;
2     $A \leftarrow (A \setminus \{C_i, C_j\}) \cup \{C_i \cup C_j\}$ ;
```

The following two lemmas and the corollary state the correctness and the optimality of Alg. 1.

Lemma 10. *Given $M \subseteq \mathbb{N}^k$, the partition A returned by* refinement*(M) is such that $\gamma_A \circ \alpha_A(M) = M$.*

Proof. Initially $A = \{\{1\},\dots,\{k\}\}$ so $\gamma_A \circ \alpha_A(M) = M$ and so $\gamma_A \circ \alpha_A(M) \subseteq M$ which is an invariant maintained by the iteration following Lem. 9. □

Lemma 11. *Given $M \subseteq \mathbb{N}^k$ and A be the partition returned by* refinement*(M). There is no partition A' with $A \preceq A'$ and $A \neq A'$ such that $\gamma_{A'} \circ \alpha_{A'}(M) = M$.*

Proof. Suppose that such a partition A' exists. Since $A \preceq A'$, $\exists C_i, C_j \in A \, \exists C' \in A' \colon (C_i \neq C_j) \wedge C_i \cup C_j \subseteq C'$. We conclude from Lem. 9 and $\gamma_{A'} \circ \alpha_{A'}(M) \subseteq M$.

By monotonicity we have that $\gamma_{A_{C_i \cup C_j}} \circ \alpha_{A_{C_i \cup C_j}}(M) \subseteq \gamma_{A_{C'}} \circ \alpha_{A_{A'}}(M) \subseteq M$. Since $M \subseteq \gamma_{A_{C_i \cup C_j}} \circ \alpha_{A_{C_i \cup C_j}}(M)$ by Galois insertion, we conclude that $\gamma_{A_{C_i \cup C_j}} \circ \alpha_{A_{C_i \cup C_j}}(M) = M$.

Hence, the condition of the while loop of the refinement algorithm is verified by A, hence the algorithm should execute at least once the loop before termination and return a partition A'' such that $A \preceq A''$ and $A \neq A''$. □

Putting together Lem. 4 and 11 we get:

Corollary 1. *Given $M \subseteq \mathbb{N}^k$, the partition A returned by* refinement(M) *is the coarsest partition such that $\gamma_A \circ \alpha_A(M) = M$.*

6 The Algorithm

The algorithm we propose is given in Alg. 2. Given a Petri net N and a $\leqslant$-dc-set S, the Algorithm builds abstract nets $\hat{N}$ with smaller dimensionality than N (line 4), analyses them (lines 5-13), and refines them (line 15) until it concludes. To analyse an abstraction $\hat{N}$, the algorithm first uses a model-checker that answers the coverability problem for $\hat{N}$ and the $\leqslant$-dc-set $\alpha_i(S)$ using any algorithm proposed in [2, 15, 14]. Besides an answer those algorithms return an overapproximation of the fixpoint $post^*_{\hat{N}}(\widehat{m_0})$ that satisfies A1–4. If the model-checker returns a positive answer then, following the abstract interpretation theory, Algorithm 2 concludes that $post^*_N(m_0) \subseteq S$ (line 6).

Otherwise, Algorithm 2 tries to decide if $\{m_0\} \not\subseteq \widetilde{pre}^*_N(S)$ checking the inclusion given by (2) (line 9-13). The fixpoint of (2) is computable ([4]) but practically difficult to build for the net N and S. Hence, our algorithm only builds an overapproximation by evaluating the fixpoint on the abstract net $\hat{N}$ instead of N, i.e. we evaluate the fixpoint $gfp\lambda X.\, \alpha_i(S) \cap \bigcap_{t\in T} \alpha_i \circ \widetilde{pre}_N[t] \circ \gamma_i(X)$ whose concretization is an overapproximation of $gfp\lambda X.\, S \cap \widetilde{pre}_N(X)$. Since the abstractions $\hat{N}$ have a smaller dimensionality than N, the greatest fixpoint can be evaluated more efficiently on $\hat{N}$. Moreover, at the ith iteration of the algorithm (i) we restrict the fixpoint to the overapproximation $\mathcal{R}_i$ of $post^*_{\hat{N}}(\alpha_i(m_0))$ computed at line 5, and (ii) we consider $\alpha_i(Z_i)$ instead of $\alpha_i(S)$. Point (i) allows the algorithm to use the information given by the forward analysis of the model-checker to obtain a smaller fixpoint, and point (ii) is motivated by the fact that at each step i we have $gfp\lambda X.\, \alpha_i(S) \cap \mathcal{R}_i \cap \bigcap_{t\in T} \alpha_i \circ \widetilde{pre}_N[t] \circ \gamma_i(X) \subseteq \alpha_i(Z_i) \subseteq \alpha_i(S)$. That allows us consider $\alpha_i(Z_i)$ instead of $\alpha_i(S)$ without changing the fixpoint, leading to a more efficient computation of it (see [4] for more details). Those optimisations are safe in the sense that the fixpoint we evaluate at line 9 does not contain $\alpha_i(m_0)$ implies that $post^*_N(m_0) \not\subseteq S$, hence its usefulness to detect negative instances (line 10).

If the algorithm cannot conclude, it refines the abstraction. The main property of the refinement is that the sequences of $Z_i's$ computed at line 9 is strictly decreasing and converge in a finite number of steps to $\widetilde{pre}^*_N(S) \cap \mathcal{R}$ where $\mathcal{R}$ is an inductive overapproximation of $post^*_N(m_0)$. Suppose that at step i, we have $Z_{i+1} = \widetilde{pre}^*_N(S) \cap \mathcal{R}$. Hence, $\gamma_{i+1} \circ \alpha_{i+1}(\widetilde{pre}^*_N(S) \cap \mathcal{R}) = \widetilde{pre}^*_N(S) \cap \mathcal{R}$. If $post^*_N(m_0) \subseteq S$ then $post^*_N(m_0) \subseteq \widetilde{pre}^*_N(S) \cap \mathcal{R}$ and the abstract interpretation theory guarantees that $post^*_{\hat{N}}(\alpha_{i+1}(m_0)) \subseteq \alpha_{i+1}(\widetilde{pre}^*_N(S) \cap \mathcal{R}) \subseteq \alpha_{i+1}(S)$, hence the model-checker will return the answer OK at iteration $i+1$. Moreover, if $post^*_N(m_0) \not\subseteq S$ then $\{m_0\} \not\subseteq \widetilde{pre}^*_N(S)$, hence $\{m_0\} \not\subseteq \widetilde{pre}^*_N(S) \cap \mathcal{R}$, and the algorithm will return KO at step $i+1$ because we have $Z_{i+1} = \widetilde{pre}^*_N(S) \cap \mathcal{R}$, hence $\{\widehat{m_0}\} \not\subseteq \alpha_{i+1}(Z_{i+1})$ by monotonicity of α_{i+1} and Z_{i+1} does not include $\{\widehat{m_0}\}$. Again, we do not evaluate the greatest fixpoint $\widetilde{pre}^*_N(S)$ because the dimensionality of N is too high and the evaluation is in general too costly in practice. Hence, we prefer to build overapproximations that can be computed more efficiently.

We now formally prove that our algorithm is sound, complete and terminates.

Lemma 12. *In Algorithm 2, for any value of i we have $post^*_N(m_0) \subseteq \gamma_i(\mathcal{R}_i)$.*

Proof. We conclude from condition A1 that $post^*_{\hat{N}}(\widehat{m_0}) \subseteq \mathcal{R}_i$, hence that $\alpha_i(post^*_N(m_0)) \subseteq \mathcal{R}_i$ by abstract interpretation and finally that $post^*_N(m_0) \subseteq \gamma_i(\mathcal{R}_i)$ by $\underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}}$. □

Proposition 4 (Soundness). *If Algorithm 2 says "OK" then we have $post^*(m_0) \subseteq S$.*

Proof. If Algorithm says "OK" then

$$\begin{array}{ll}
\mathcal{R}_i \subseteq \alpha_i(S) & \\
\Rightarrow \gamma_i(\mathcal{R}_i) \subseteq \gamma_i \circ \alpha_i(S) & \gamma_i \text{ is monotonic} \\
\Rightarrow \gamma_i(\mathcal{R}_i) \subseteq S & \text{Line 2,15 and Lem. 10} \\
\Rightarrow post^*_N(m_0) \subseteq S & \text{Lem. 12}
\end{array}$$

□

Algorithm 2. Algorithm for the coverability problem, assume $\{m_0\} \subseteq S$

```
   Data: A Petri net $N = (P, T, F, m_0)$, a $\leqslant$-dc-set $S$
1  $Z_0 = S$
2  $A_0 = \mathsf{refinement}(Z_0)$
3  for $i = 0, 1, 2, 3, \ldots$ do
4      $\boxed{\textbf{Abstract:}}$ Given $A_i$, compute $\hat{N}$ given by def. 8.
5      $\boxed{\textbf{Verify:}}$ $(answer, \mathcal{R}_i) = \mathsf{Checker}(\widehat{m_0}, post_{\hat{N}}, \alpha_i(S))$
6      if $answer == OK$ then
7          return $OK$
8      else
9          Let $\mathcal{S}_i = gfp\lambda X.\, \alpha_i(Z_i) \cap \mathcal{R}_i \cap \bigcap_{t \in T} \alpha_i \circ \widetilde{pre}_N[t] \circ \gamma_i(X)$
10         if $\widehat{m_0} \not\subseteq \mathcal{S}_i$ then
11             return $KO$
12         end
13     end
14     Let $Z_{i+1} = Z_i \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(\gamma_i(\mathcal{S}_i))$
15     $\boxed{\textbf{Refine:}}$ Let $A_{i+1} = A_i \curlywedge \mathsf{refinement}(Z_{i+1})$
16 end
```

We need intermediary results (Prop. 5 and Lem. 13) to establish the completeness of Algorithm 2. The following result is about greatest fixpoints.

Proposition 5. *Let $S, R, \widehat{R} \subseteq \mathbb{N}^k$ such that $\widehat{R} \subseteq R$, $post(R) \subseteq R$ and $post(\widehat{R}) \subseteq \widehat{R}$.*

$$gfp\lambda X.\,(S \cap \widehat{R} \cap \widetilde{pre}(X)) = \widehat{R} \cap gfp\lambda X.\,(S \cap R \cap \widetilde{pre}(X)) \ .$$

Lemma 13. *In Alg. 2 if $post_N^*(m_0) \subseteq S$ then for any i we have $post_N^*(m_0) \subseteq Z_i$.*

Proof. The proof is by induction on i.
base case. Trivial since line 1 defines Z_0 to be S.
inductive case. Line 9 shows that $\gamma_i(\mathcal{S}_i)$ overapproximates $gfp\lambda X.\, Z_i \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(X)$, hence that $Z_{i+1} \supseteq gfp\lambda X.\, Z_i \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(X)$ by line 14.

$$\begin{array}{lr}
post_N^*(m_0) \subseteq Z_i & \text{hyp} \\
\Rightarrow post_N^*(m_0) \subseteq gfp\lambda X.\, Z_i \cap \widetilde{pre}_N(X) & \text{def of } post_N^*(m_0)\text{, (Gc) p. 127} \\
\Rightarrow post_N^*(m_0) \subseteq \gamma_i(\mathcal{R}_i) \cap gfp\lambda X.\, Z_i \cap \widetilde{pre}_N(X) & \text{Lem. 12} \\
\Leftrightarrow post_N^*(m_0) \subseteq gfp\lambda X.\, Z_i \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(X) & \text{Prop. 5, } post_N(\gamma_i(\mathcal{R}_i)) \subseteq \gamma_i(\mathcal{R}_i) \\
\Rightarrow post_N^*(m_0) \subseteq Z_{i+1} & \text{by above}
\end{array}$$

□

Proposition 6 (Completeness). *If Alg. 2 says "KO" then we have $post_N^*(m_0) \subseteq S$.*

Proof. If Algorithm says "KO" then

$$\begin{aligned}
&\widehat{m_0} \not\subseteq \mathcal{S}_i \\
&\Leftrightarrow \alpha(m_0) \not\subseteq \mathcal{S}_i && \text{def of } \widehat{m_0} \\
&\Leftrightarrow m_0 \not\subseteq \gamma_i(\mathcal{S}_i) && \overset{\gamma_i}{\underset{\alpha_i}{\leftrightarrows}} \\
&\Leftrightarrow m_0 \not\subseteq \gamma_i(\mathit{gfp}\lambda X.\, \alpha_i(Z_i) \cap \mathcal{R}_i \cap \bigcap_{t \in T} \alpha_i \circ \widetilde{pre}_N[t] \circ \gamma_i(X)) && \text{def of } \mathcal{S}_i \\
&\Rightarrow m_0 \not\subseteq \gamma_i(\mathit{gfp}\lambda X.\, \alpha_i(Z_i) \cap \alpha_i \circ \gamma_i(\mathcal{R}_i) \cap \bigcap_{t \in T} \alpha_i \circ \widetilde{pre}_N[t] \circ \gamma_i(X)) && \overset{\gamma_i}{\underset{\alpha_i}{\leftrightarrows}} \\
&\Rightarrow m_0 \not\subseteq \gamma_i(\mathit{gfp}\lambda X.\, \alpha_i(Z_i \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(\gamma_i(X)))) && \alpha(\cap\cdot) \subseteq \cap\alpha(\cdot) \\
&\Rightarrow m_0 \not\subseteq \mathit{gfp}\lambda X.\, Z_i \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(X) && \overset{\gamma_i}{\underset{\alpha_i}{\leftrightarrows}} \\
&\Leftrightarrow \mathit{lfp}\lambda X.\, m_0 \cup \mathit{post}(X) \not\subseteq Z_i \cap \gamma_i(\mathcal{R}_i) && (1) \equiv (2) \\
&\Rightarrow \mathit{lfp}\lambda X.\, m_0 \cup \mathit{post}(X) \not\subseteq Z_i && \text{Lem. 12} \\
&\Rightarrow \mathit{lfp}\lambda X.\, m_0 \cup \mathit{post}(X) \not\subseteq S && \text{Lem. 13}
\end{aligned}$$

□

Proposition 7 (Termination). *Algorithm 2 terminates.*

Proof. It is clear by line 14 that we have

$$Z_0 \supseteq Z_1 \supseteq \cdots \supseteq Z_i \supseteq Z_{i+1} \supseteq \cdots$$

Consider the sequence of Z_i's and assume that from index i we have $Z_{i+1} = Z_i$.

$$\begin{aligned}
&Z_{i+1} = Z_i \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(\gamma_i(\mathcal{S}_i)) && \text{def of } Z_{i+1} \\
&\Rightarrow Z_{i+1} \subseteq S \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(\gamma_i(\mathcal{S}_i)) && Z_i \subseteq Z_0 = S \\
&\Rightarrow Z_{i+1} \subseteq S \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(Z_i) && \widetilde{pre}_N \text{ is monotonic,} \gamma_i(\mathcal{S}_i) \subseteq Z_i \\
&\Leftrightarrow Z_{i+1} \subseteq S \cap \gamma_i(\mathcal{R}_i) \cap \widetilde{pre}_N(Z_{i+1}) && Z_{i+1} = Z_i \\
&\Leftrightarrow Z_j \subseteq S \cap \gamma_{j-1}(\mathcal{R}_{j-1}) \cap \widetilde{pre}_N(Z_j) && \text{let } j = i+1 \\
&\Rightarrow Z_j \subseteq S \wedge Z_j \subseteq \widetilde{pre}_N(Z_j) && \text{glb} \\
&\Leftrightarrow Z_j \subseteq S \wedge \mathit{post}_N(Z_j) \subseteq Z_j && \text{(Gc)} \\
&\Rightarrow \alpha_j(Z_j) \subseteq \alpha_j(S) \wedge \alpha_j \circ \mathit{post}_N(Z_j) \subseteq \alpha_j(Z_j) && \alpha_j \text{ is monotonic} \\
&\Rightarrow \alpha_j(Z_j) \subseteq \alpha_j(S) \wedge \alpha_j \circ \mathit{post}_N(\gamma_j \circ \alpha_j(Z_j)) \subseteq \alpha_j(Z_j) && \text{line 15, Lem. 10} \\
&\Rightarrow \alpha_j(Z_j) \subseteq \alpha_j(S) \wedge \mathit{post}_{\hat{N}}(\alpha_j(Z_j)) \subseteq \alpha_j(Z_j) && \text{Prop. 2}
\end{aligned}$$

Then, either $\widehat{m_0} \subseteq \alpha_j(Z_j)$ and so [12, Thm. 4] shows that

$$\begin{aligned}
&\mathit{lfp}\lambda X.\, \widehat{m_0} \cup \mathit{post}_{\hat{N}}(X) \subseteq \alpha_j(S) \\
&\Rightarrow \gamma_j(\mathit{lfp}\lambda X.\, \widehat{m_0} \cup \mathit{post}_{\hat{N}}(X)) \subseteq \gamma_j \circ \alpha_j(S) \\
&\Rightarrow \gamma_j(\mathit{lfp}\lambda X.\, \widehat{m_0} \cup \mathit{post}_{\hat{N}}(X)) \subseteq S && \text{by line 2,15 and Lem. 3}
\end{aligned}$$

and line 6 shows that the algorithm terminates. Or we have,

$$\begin{aligned}
&\widehat{m_0} \not\subseteq \alpha_j(Z_j) \\
&\Rightarrow \widehat{m_0} \not\subseteq S_j && S_j \subseteq \alpha_j(Z_j) \text{ by line 9}
\end{aligned}$$

and line 10 shows that the algorithm terminates.

Table 1. Var: number of places of the Petri net; **Cvar**: number of places of the abstraction that allow to conclude; **Ref**: number of refinements before conclusion; **time**: execution time in seconds on Intel Xeon 3Ghz

	Example	Var	Cvar	Ref	time
Unbounded PN	ME	5	4	3	0.02
	multiME	12	5	3	2.69
	FMS	22	4	3	8.30
	CSM	14	9	4	11.78
	mesh2x2	32	9	4	340
	mesh3x2	52	9	4	3357
Bounded PN	**Example**	**Var**	**Cvar**	**Ref**	**time**
	lamport	11	9	4	8.50
	dekker	16	15	4	60.2
	peterson	14	12	5	21.5

Now we assume that the sequence of Z_i's strictly decreases, i.e. $Z_{i+1} \subset Z_i$. First recall that the ordered set $\langle \subseteq, DCS(\mathbb{N}^k) \rangle$ is a wqo. We conclude from A2, Lem. 5, $\leqslant$-dc-set are closed to $\widetilde{pre}$ and $\cap$ that for any value of i in Alg. 2 we have $Z_i \in DCS(\mathbb{N}^k)$. However $\leqslant$ defines a wqo and following [14, Lem. 2] there is no infinite strictly decreasing sequence of $\langle \subseteq, DCS(\mathbb{N}^k) \rangle$, hence a contradiction. $\square$

7 Experimental Results

We implemented Alg. 2 in C using the symbolic data structure of [17] to represent and manipulate sets of markings. We used, for the model-checker referenced at line 5, the algorithm of [15].

We tested our method against a large set of examples. The properties we consider are mutual exclusions and the results we obtained are shown in Table 1. We distinguish two kind of examples. *Parameterized systems* describe systems where we have a parameterized number of resources: ME [5, Fig. 1], MultiME (Fig. 1 of Sect. 2), CSM [18, Fig. 76, page 154], FMS [19], the mesh 2x2 of [18, Fig. 130, page 256] and its extension to the 3x2 case. For all those infinite state Petri nets, the mutual exclusion properties depend only on a small part of the nets.

The mesh 2x2 (resp. 3x2) examples corresponds to 4 (resp. 6) processors running in parallel with a load balancing mechanism that allow tasks to move from one processor to another. The mutual exclusion property says that one processor never processes two tasks at the same time. That property is local to one processor and our algorithm builds an abstraction where the behaviour of the processor we consider is exactly described and the other places are totally abstracted into one place. In that case, we manipulate subsets of $\mathbb{N}^9$ instead of subsets of $\mathbb{N}^{32}$ for mesh 2x2 or $\mathbb{N}^{52}$ for mesh 3x2.

For the other examples, we have a similar phenomenon: only a small part of the Petri nets is relevant to prove the mutual exclusion property. The rest of the net describes other aspects of the parameterized system and is abstracted by our algorithm. Hence, all the parameterized systems are analysed building an abstract Petri net with few places.

The bounded Petri Net examples are classical *algorithms to ensure mutual exclusion* of critical sections for two processes. In those cases, our method concludes building very precise abstractions, i.e. only few places are merged. The reasons are twofold: (i) the algorithms are completely dedicated to mutual exclusion, and (ii) the nets have been designed by hand in a "*optimal*" manner. However and quite surprisingly, we noticed that our algorithm found for those examples places that can be merged. In our opinion, this shows that our algorithm found reductions that are (too) difficult to find by hand.

Execution Times and Future Improvements. For all the examples we considered, the execuion times of the checker [15] on the abstract Petri nets that allows Algorithm 2 to conclude are smaller than the execution times of the checker on the concrete Petri nets, showing the interest of reducing the dimentionality of Petri nets before verification. For instance, the execution time of the checker on the concrete Petri nets of the mesh2x2 is 1190 seconds, and the mesh3x2 is greater than 5 hours (on Intel Xeon 3Ghz). Hence, for those examples Algorithm 2 is much more efficient than directly check the concrete Petri net. We also noticed that our prototype spends most of its time in the refinement step. We currently use a naive implementation of Algorithm 1 that can be greatly improved. As a consequence, some research effort are needed to define efficient techniques leading to reasonable cost for refinement of Petri nets.

References

1. German, S.M., Sistla, A.P.: Reasoning about Systems with Many Processes. Journal of ACM 39(3), 675–735 [1] (1992) [124]
2. Karp, R.M., Miller, R.E.: Parallel program schemata. Journal of Comput. Syst. Sci. 3(2), 147–195 [1, 5, 15] (1969) [124, 128, 137]
3. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere? Theoretical Computer Science 256(1-2), 63–92 [1] (2001) [124]
4. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proc. of the 11th Annual IEEE Symp. on Logic in Computer Science (LICS), pp. 313–321. IEEE Computer Society Press, Washington [1, 6, 15] (1996) [124, 128, 138]
5. Delzanno, G., Raskin, J.F., Van Begin, L.: Attacking Symbolic State Explosion. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 298–310. Springer, Heidelberg [1, 18] (2001) [124, 141]
6. Van Begin, L.: Efficient Verification of Counting Abstractions for Parametric systems. PhD thesis, Université Libre de Bruxelles, Belgium [1] (2003) [124]
7. Abdulla, P.A., Iyer, S.P., Nylén, A.: Unfoldings of unbounded petri nets. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 495–507. Springer, Heidelberg [1] (2000) [124]
8. Grahlmann, B.: The PEP Tool. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 440–443. Springer, Heidelberg [1] (1997) [124]
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 [3] (2003) [126]

10. Berthelot, G., Roucairol, G., Valk, R.: Reductions of nets and parallel prgrams. In: Advanced Cource: Net Theory and Applications. LNCS, vol. 84, pp. 277–290. Springer, Heidelberg [3] (1975) [126]
11. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 77: Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, pp. 238–252. ACM Press, New york [2, 7, 8] (1977) [125, 130, 131]
12. Cousot, P.: Partial completeness of abstract fixpoint checking, invited paper. In: Choueiry, B.Y., Walsh, T. (eds.) SARA 2000. LNCS (LNAI), vol. 1864, pp. 1–25. Springer, Heidelberg [4, 5, 18] (2000) [127, 128, 140]
13. Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. Amer. J. Math. 35, 413–422 [5] (1913) [128]
14. Ganty, P., Raskin, J.F., Van Begin, L.: A complete abstract interpretation framework for coverability properties of WSTS. In: VMCAI 2006. LNCS, vol. 3855, pp. 49–64. Springer, Heidelberg [128, 138, 141] (2006) [128, 137, 141]
15. Geeraerts, G., Raskin, J.F., Van Begin, L.: Expand, enlarge and check: new algorithms for the coverability problem of WSTS. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 287–298. Springer, Heidelberg [5, 15, 18, 19] (2004) [128, 137, 141, 142]
16. Burris, S., Sankappanavar, H.P.: A Course in Universal Algebra. Springer, New York [6, 9] (1981) [129, 132]
17. Ganty, P., Meuter, C., Delzanno, G., Kalyon, G., Raskin, J.F., Van Begin, L.: Symbolic data structure for sets of k-uples. Technical report [18] (2007) [141]
18. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. Series in Parallel Computing [18] (1995) [141]
19. Ciardo, G., Miner, A.: Storage Alternatives for Large Structured State Space. In: Marie, R., Plateau, B., Calzarossa, M.C., Rubino, G.J. (eds.) Computer Performance Evaluation Modelling Techniques and Tools. LNCS, vol. 1245, pp. 44–57. Springer, Heidelberg [18] (1997) [141]

A Compositional Method for the Synthesis of Asynchronous Communication Mechanisms

Kyller Gorgônio[1], Jordi Cortadella[2], and Fei Xia[3]

[1] Embedded Systems and Pervasive Computing Laboratory
Federal University of Campina Grande, Brazil
[2] Department of Software
Universitat Politècnica de Catalunya, Spain
[3] School of Electrical, Electronic and Computer Engineering
University of Newcastle upon Tyne, UK

Abstract. Asynchronous data communication mechanisms (ACMs) have been extensively studied as data connectors between independently timed concurrent processes. In previous work, an automatic ACM synthesis method based on the generation of the reachability graph and the theory of regions was proposed. In this paper, we propose a new synthesis method based on the composition of Petri net modules, avoiding the exploration of the reachability graph. The behavior of ACMs is formally defined and correctness properties are specified in CTL. Model checking is used to verify the correctness of the Petri net models. The algorithms to generate the Petri net models are presented. Finally, a method to automatically generate C++ source code from the Petri net model is described.

Keywords: Asynchronous communication mechanisms, Petri nets, concurrent systems, synthesis, model checking, protocols.

1 Introduction

One of the most important issues when designing communication schemes between asynchronous processes is to ensure that such schemes allow as much asynchrony as possible after satisfying design requirements on data. When the size of computation networks becomes large, and the traffic between the processing elements increases, this task becomes more difficult.

An *Asynchronous Communication Mechanism* (ACM) is a scheme which manages the transfer of data between two processes, a *producer* (writer) and a *consumer* (reader), not necessarily synchronized for the purpose of data transfer. The general scheme of an ACM is shown in Figure 1. It includes a shared memory to hold the transferred data and control variables. In this work it is assumed that the data being transferred consists of a stream of items of the same type, and the writer and reader processes are single-threaded loops. At each iteration a single data item is transferred to or from the ACM.

Classical semaphores can be configured to preserve the coherence of write and read operations. However, this approach is not satisfactory when data items are large and a minimum locking between the writer and the reader is expected [4].

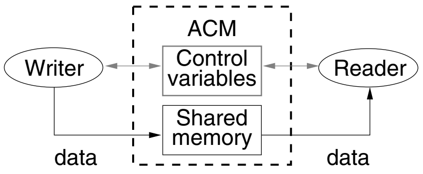


Fig. 1. ACM with shared memory and control variables

By using single-bit unidirectional variables, the synchronization control can be reduced to the reading and writing of these variables by extremely simple atomic actions [6]. Variables are said to be *unidirectional* when they can only be modified by one of the processes. This provides the safest solution for a maximum asynchrony between the writer and the reader. In particular, if the setting, resetting and referencing of control variables can be regarded as atomic events, the correctness of ACMs becomes easy to prove.

ACMs are classified according to their *overwriting* and *re-reading* policies [8,6]. Overwriting occurs when the ACM is full of data that has not been read before. In this case the producer can overwrite some of the existing data items in the buffer. Re-reading occurs when all data in the ACM has been read by the consumer. In this case the consumer is allowed to re-read an existing item. Table 1 shows such a classification. `BB` stands for a bounded buffer that does not allow neither overwriting nor re-reading. `RRBB` stands for an ACM is that only allows re-reading. On the other hand, the `OWBB` scheme allows only overwriting. Finally, the `OWRRBB` scheme allows both re-reading and overwriting.

Table 1. Classification of ACMs

	No re-reading	**Re-reading**
No overwriting	`BB`	`RRBB`
Overwriting	`OWBB`	`OWRRBB`

The choice of using a particular class of ACM for a particular job is generally based on data requirements and system timing restrictions [4,6]. For the re-reading ACM class, it is more convenient to re-read the item from the previous cycle rather than an item from several cycles before. For overwriting, the typical cases consist of overwriting either the newest or the oldest item in the buffer [6,9,2]. Overwriting the newest item in the buffer [9] attempts to provide the reader with the best continuity of data items for its next read. Continuity is one of the primary reasons for having a buffer of significant size. Overwriting the oldest item is based on the assumption that newer data is always more relevant than older.

1.1 ACM Example

Now consider an RRBB ACM with three data cells. The single-bit (boolean) control variables r_i and w_i, with $i \in \{1, 2, 3\}$, are used to indicate which cell each process must access. Initially the reader is pointing at cell 0, $r_0 = 1$ and $r_1 = r_2 = 0$, and the writer to cell 1, $w_1 = 1$ and $w_0 = w_2 = 0$. The shared memory is initialized with some data. This scheme is shown in Figure 2.

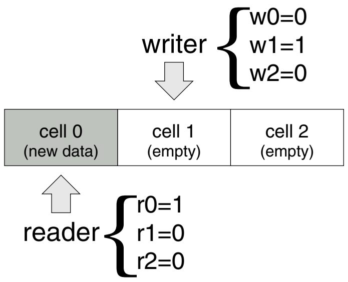


Fig. 2. Execution of RRBB ACM with 3 cells

The writer always stores some data into the ACM and then attempts to advance to the next cell releasing the new data. In this way, a possible trace for the writer is $\langle wr_1 wr_2 wr_0 wr_1 \rangle$, where wr_i denotes "*write data on cell i*". A similar behavior applies to the reader. A possible trace for the reader is $\langle rd_0 rd_1 rd_1 rd_2 \rangle$.

In an RRBB ACM, no overwriting and allowing re-reading imply the following behavior:

– The writer first accesses the shared memory and then advances to the next cell, but only if the reader is not pointing at it.
– The reader first advances to the next cell if the writer is not there and then performs the data transfer, otherwise it re-reads the current cell.

In general, and depending on how the read and write traces interleave, coherence and freshness properties must be satisfied.

Coherence is related to mutual exclusion between the writer and the reader. For example, a possible trace for this system is $\langle wr_1 wr_2 rd_0 \cdots \rangle$. After the writer executing twice, the next possible action for both processes is to access cell 0. This introduces the problem of data coherence when the reader and the writer are retrieving and storing data on the same memory locations.

Freshness is related to the fact that the last data record produced by the writer must be available for the reader. On the ACMs studied in this work, the reader always attempts to retrieve the oldest data stored in the shared memory that has not been read before. This means that the freshness property imposes a specific sequencing of data, i.e. the data is read in the same order that it is written. Depending on the ACM class, some data may be read more than once or may be missed. However, the sequence should be preserved. For the example above, one possible trace is $\langle wr_1 rd_0 wr_2 rd_1 rd_1 \cdots \rangle$. Note that at the moment the reader executes the first rd_1 action, the writer has already executed a wr_2. This means that there is some new data on cell 2. But the reader is engaged to execute rd_1 again, which violates freshness.

With a correct interleaving both processes will avoid accessing the same data cell at the same time, the writer will not be allowed to overwrite unread data, and the reader will have the possibility of re-reading the most recent data only when there is no unread data in the ACM. For the example above, a correct trace is $\langle wr_1 rd_0 rd_1 wr_2 rd_1 wr_0 rd_2 wr_1 \rangle$. Observe that the sub-trace $rd_1 wr_2 rd_1$ does not contradict the fact that the reader only re-reads any data if there is no new one available. This is because after the first rd_1 there is no new data, then the reader prepares to re-read and from this point it will engage on a re-reading regardless the actions of the writer.

Algorithm 1. RRBB ACM with 3 cells

```
**Require:** Boolean $w_0, w_1, w_2$
**Require:** External Boolean $r_0, r_1, r_2$
**process** writer()
    $w_1 := 1$; $w_0 := w_2 := 0$;
    **loop**
        **if** $w_0 = 1 \land r_1 = 0$ **then**
            write cell 1;
            $w_0 := 0$; $w_1 := 1$;
        **else if** $w_1 = 1 \land r_2 = 0$ **then**
            write cell 2;
            $w_1 := 0$; $w_2 := 1$;
        **else if** $w_2 = 1 \land r_0 = 0$ **then**
            write cell 0;
            $w_2 := 0$; $w_0 := 1$;
        **else**
            wait until some $r_i$ is modified;
        **end if**
    **end loop**
**end process**
```

```
**Require:** Boolean $r_0, r_1, r_2$
**Require:** External Boolean $w_0, w_1, w_2$
**process** reader()
    $r_0 := 1$; $r_1 := r_2 := 0$;
    **loop**
        **if** $r_0 = 1 \land w_1 = 0$ **then**
            $r_0 := 0$; $r_1 := 1$;
            read cell 1;
        **else if** $r_0 = 1 \land w_1 = 1$ **then**
            read cell 0;
        **else if** $r_1 = 1 \land w_2 = 0$ **then**
            $r_1 := 0$; $r_2 := 1$;
            read cell 2;
        **else if** $r_0 = 1 \land w_1 = 1$ **then**
            read cell 1;
        **else if** $r_2 = 1 \land w_0 = 0$ **then**
            $r_2 := 0$; $r_0 := 1$;
            read cell 0;
        **else if** $r_2 = 1 \land w_0 = 1$ **then**
            read cell 2;
        **end if**
    **end loop**
**end process**
```

A possible implementation of the example above is described in Algorithm 1. The writer is shown on the left side and the reader on the right. Each process consists of an infinite loop. This is just a simple abstraction of the real behavior of a process, in which the ACM operations are combined with the data processing actions. At each ACM operation:

– The writer first writes to the shared memory and then tries to advance to the next cell by modifying its control variable w, if this is contradictory to the current values of the reader's control variable r, the writer waits. Note that when the writer is waiting, the data item just written into the ACM is not available for the reader to read because the writer has not yet completed its move to the next cell.

– The reader first tries to advance to the next cell by modifying its control variable r, if this is contradictory to the current values of the writer's control variable w, no modification to r occurs, in either case (with or without successfully advancing) the reader then reads (or rereads) from cell r. Note that cell r cannot be accessed by the writer, even if its content has already been read by the reader.

In other words, at any time, each of the writer and reader processes "owns" a cell, and for data coherence purposes any cell can only "belong to" one of these processes at any time. Furthermore, since only binary control variables are used, the size of this description grows with the size of the ACM. This means that more variables are needed, and for overwriting ACM classes it is more difficult to correctly deal with all of them.

In the rest of this paper, a Petri net based method for the automatic synthesis of ACMs is presented. The method receives as input a functional specification consisting of the ACM class that should be implemented by the ACM and the number of cells it should have. As output, it produces the source code implementing the operations with the ACM. The implementation can be either in software (e.g. C++ or Java) or hardware (e.g. Verilog or VHDL).

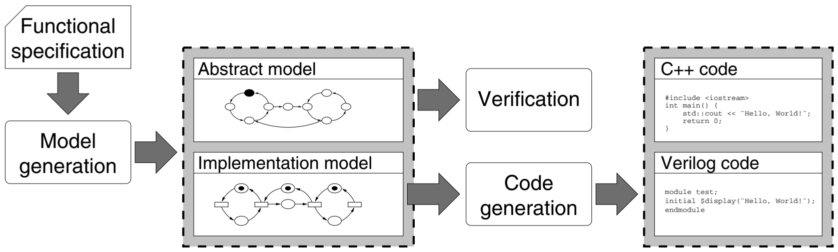
In this paper, we will provide C++ implementations. For instance, the C++ code for the 3-cell RRBB ACM described above is shown in Figures 6 and 7.

In the next sections, the methodology presented in the paper will be described. The behavior of the RRBB ACM class will be formally defined and the method to generate its implementation will be detailed. Due to the limited space, the OWBB and OWRRBB classes will not be discussed in detail. However, the principle used to generate the RRBB implementations also applies to the overwriting ACM classes.

2 Overview of the Approach

In previous work [1,8,10], a step-by-step method based on the theory of regions for the synthesis of ACMs was presented. The method required the generation of the complete state space of the ACM by exploring all possible interleavings between the reader and the writer actions. The state space of the ACM was generated from its functional specification. Next, a Petri net model was obtained using the concept of ACM regions, a refined version of the conventional regions.

This work proposes the generation of the Petri net model using a modular approach that does not require the explicit enumeration of the state space. The Petri net model is build by abutting a set of Petri net modules. The correctness of the model can then be formally verified using model checking. The relevant properties of the ACM, coherence and freshness, can be specified using CTL formulae. This paper also extends previous work by introducing an approach to automatically generate the implementation of the ACM from the Petri net model. Figure 3 shows the design flow for the automatic generation of ACMs.



```
#include <iostream>
int main() {
    std::cout << "Hello, World!";
    return 0;
}
```

```
module test;
initial $display("Hello, World!");
endmodule
```

Fig. 3. The design flow

Compared to the previous work, the new approach has the advantage of not dealing with the entire state space of the ACM when generating the Petri net model. It is obtained in linear time. On the other hand, it requires to verify the model generated to provide enough evidence of its correctness. Observe that it is possible to obtain the ACM implementation without doing verification. In practice, the new approach allows to obtain the Petri net model when the size of the ACM grows.

2.1 Models for Verification and Implementation

The two basic paradigms on the approach presented in this paper are *automation* and *correctness*. For that reason, from the functional specification of an ACM, two formal models are generated:

– An *abstract model*, that describes the possible traces of the system and that is suitable for model checking of the main properties of the ACM: coherence and freshness. These properties can be modeled using temporal logic formulae.
– An *implementation model*, that is suitable for generating a hardware or software implementation of the ACM. This model is generated by the composition of basic Petri net modules and contains more details about the system. This model is required to narrow the distance between the behavior and the implementation.

For a complete verification of the system, a bridge is required to check that the implementation model is a refinement of the abstract model. For such purpose, the Cadence SMV Model Checker [5] has been used.

The Cadence SMV extends the CMU SMV model checker by providing a more expressive description language and by supporting a variety of techniques for compositional verification. In particular, it supports refinement verification by allowing the designer to specify many abstract definitions for the same signal. It can then check if the signal in a more abstract level is correctly implemented by another abstraction of lower level.

Thus, the correctness of the generated ACMs is verified as follows:

1. The abstract and implementation models of the ACM are generated.
2. The properties of the ACM are specified in CTL and model checked on the abstract model.

3. The implementation model is verified to be a refinement of the abstract model.

In the forthcoming section, the abstract and implementation models for the class of RRBB ACMs are presented.

3 The Abstract Model for RRBB ACMs

The abstract model for an RRBB ACM is specified as a transition system. The state of the ACM is defined by the data items available for reading. For each state, σ defines the queue of data stored in the ACM. More specifically, σ is a sequence: $\sigma = a_0 a_1 \cdots a_{j-1} a_j$, with $j < n$, where n is size of the ACM. a_j is the last written data, and a_0 is the next data to be retrieved by the reader. The size of the ACM is given by its number of cells, i.e. the maximum number of data items the ACM can store at a certain time.

σ must also express if the processes are accessing the ACM or not. This is done by adding flags to the a_0 and a_j items. a_j^w indicates that the writer is producing data a_j, and this data is not yet available for reading. Similarly, a_0^r is used to indicate that the reader is consuming data a_0.

Observe that σ can be interpreted as a stream of data that is passed from the writer (on the left) to the reader (on the right). There are four events that change the state of the ACM:

- $rd_b(a)$: reading data item a begins.
- $rd_e(a)$: reading data item a ends.
- $wr_b(a)$: writing data item a begins.
- $wr_e(a)$: writing data item a ends.

The notation $\langle\sigma_i\rangle \xrightarrow{e} \langle\sigma_j\rangle$ denotes the occurrence of event e from state $\langle\sigma_i\rangle$ to state $\langle\sigma_j\rangle$, whereas $\langle\sigma\rangle \xrightarrow{e} \perp$ is used to denote that e is not enabled in $\langle\sigma\rangle$.

In RRBB ACMs, the reader is required not to wait when starting an access to the ACM. In the case there is no new data in the ACM, the reader will re-read some data that was read before.

The writer can add data in the ACM until it is full. In such case, the writer is required to wait until the reader retrieves some data from the ACM. The reader always tries to retrieve the oldest non-read data and, if all data in the ACM has been read before, then it attempts to re-read the last retrieved data item.

Definition 1 formally captures the behavior of RRBB ACMs. Rules 1-3 model the behavior of the writer. Rules 4-7 model the behavior of the reader.

Definition 1 (RRBB transition rules). *The behavior of an RRBB ACM is defined by the following set of transitions (n is the number of cells of the ACM and the cells are numbered from* 0 *to* $n-1$*):*

1. $\langle\sigma\rangle \xrightarrow{wr_b(a)} \langle\sigma a^w\rangle$ **if** $|\sigma| < n$
2. $\langle\sigma\rangle \xrightarrow{wr_b(a)} \perp$ **if** $|\sigma| = n$
3. $\langle\sigma a^w\rangle \xrightarrow{wr_e(a)} \langle\sigma a\rangle$
4. $\langle a\sigma\rangle \xrightarrow{rd_b(a)} \langle a^r\sigma\rangle$
5. $\langle a^r\sigma\rangle \xrightarrow{rd_e(a)} \langle\sigma\rangle$ **if** $|\sigma| > 0 \wedge \sigma \neq b^w$
6. $\langle a^r\rangle \xrightarrow{rd_e(a)} \langle a\rangle$
7. $\langle a^r b^w\rangle \xrightarrow{rd_e(a)} \langle ab^w\rangle$

Rule 1 models the start of a write action for a new data item a and signaling that it is not available for reading (a^w). Rule 3 models the completion of the write action and making the new data available for reading. Finally, rule 2 represents the blocking of the writer when the ACM is full ($|\sigma| = n$).

Rule 4 models the beginning of a read action retrieving data item a and indicating that it is being read (a^r). Rule 5 models the completion of the read operation. In this rule, a is removed from the buffer when other data is available. On the other hand, rules 6 and 7 model the completion of the read action when no more data is available for reading. In this case, the data is not removed from the buffer and is available for re-reading. This is necessary due to the fact that the reader is required not to be blocked even if there is no new data in the ACM.

It is important to observe that in the state $\langle a^r b^w\rangle$ the next element to be retrieved by the reader will depend on the order that events $wr_e(b)$ and $rd_e(a)$ occur. If the writer delivers b before the reader finishes retrieving a, then b will be the next data to be read. Otherwise, the reader will prepare to re-read a.

Definition 1 was modeled using the Cadence SMV model checker and freshness and coherence properties were verified. Each process was modeled as an SMV module. In the SMV language, a module is a set of definitions, such as type declarations and assignments, that can be reused. Specifically, each process consists of a `case` statement in which each condition corresponds to a rule in Definition 1. The SMV model obtained from Definition 1 will be used in Section 4 to verify a lower level specification of the ACM. Next, the specification of the coherence and freshness properties is discussed.

3.1 Coherence

To verify the coherence property it is necessary to prove that there is no reachable state in the system in which both processes are addressing the same segment of the shared memory.

In the ACM model described by Definition 1, the reader always addresses the data stored in the first position of the ACM, represented by σ. On the other hand, the writer always addresses the tail of the ACM. To prove coherence in this model it is only necessary to prove that every time the reader is accessing the ACM, then:

– it is addressing the first data item, and
– if the writer is also accessing the ACM, then it is not writing in the first location.

In other words, if at a certain time the shared memory contains a sequence of data $\sigma = a_0 a_1 \cdots a_{j-1} a_j$, with $j < n$, where n is the size of the ACM. Then:

$$\mathtt{AG}\ (a^r \in \sigma \rightarrow (a^r = a_0 \wedge (a^w \in \sigma \rightarrow a^w = a_j \wedge j > 0)))$$

The formula above specifies that for any reachable state of the system ($\mathtt{AG}$), if the reader is accessing the ACM, then:

1. It is reading a data from the beginning of the buffer ($a^r = a_0$);
2. If the writer is also accessing the ACM, then it is not pointing at the beginning of the queue ($(a^w \in \sigma \rightarrow a^w = a_j \wedge j > 0)$).

3.2 Freshness

As discussed before, freshness is related to sequencing of data. Now, let us assume that at a certain time the shared memory contains a sequence of data $\sigma = a_0 a_1 \cdots a_{j-1} a_j$, with $j < n$, a_j is the last written data, and a_0 is the next data to be retrieved by the reader. Then, at the next cycle the ACM will contain a sequence of data σ' such that one of the following is true:

1. $\sigma' = \sigma$: in this case neither the reader has removed any data item from the head of σ nor the writer has stored a new item in its tail;
2. $\sigma' = a_0 a_1 \cdots a_{j-1} a_j a_{j+1}$: in this case the reader has not removed any item from the head of σ, but the writer has added a new item to the tail;
3. $\sigma' = a_1 \cdots a_{j-1} a_j$: and, finally, in this case the reader has removed a data item from the head of σ.

The above can be specified by the following CTL formula:

$$\mathtt{AG}(|\sigma| = x \rightarrow \mathtt{AX}((|\sigma'| >= x \wedge \sigma' = \sigma^+) \vee (|\sigma'| = x - 1 \wedge \sigma' = \sigma^-)))$$

where σ^+ is used to denote $a_0 a_1 \cdots a_{j-1} a_j$ or $a_0 a_1 \cdots a_{j-1} a_j a_{j+1}$ and σ^- is used to denote $a_1 \cdots a_{j-1} a_j$. Observe that 1 and 2 are captured by the same same CTL sub-formula, which is given by the left side of the $\vee$ inside the $\mathtt{AX}$ operator.

The guidelines introduced above can be used to generate an SMV model for any RRBB ACM with three or more data cells. After that, the model can be verified against the CTL formulas for coherence and freshness. Observe that the number of CTL formulas needed to specify freshness grows linearly with the size of the ACM. This is because, for each possible size of σ, it is necessary to generate another CTL formula.

4 The Implementation Model and Its Verification

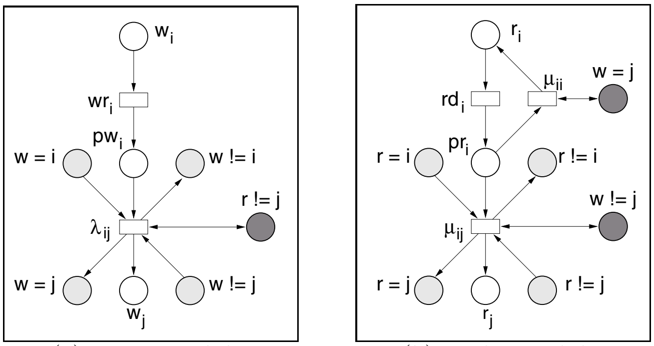
The modular approach for the generation of ACMs is now introduced by means of an example, including the generation of a Petri net implementation model and its verification.

4.1 Generation of the Implementation Model

A Petri net model for a 3-cell RRBB ACM will be generated and mapped into a C++ implementation. As stated before, this new modular approach is based on the definition of a set of elementary building blocks that can be easily assembled to construct the entire system.

The repetitive behavior of the writer consists of writing data into the i^{th} cell, checking if the reader process is addressing the next cell and, in the negative case advancing to it, otherwise waiting until the reader advances. In a similar way, the reader is expected to retrieve data from the i^{th} cell, check if the writer is accessing the next cell and, in the negative case advancing to it, otherwise preparing to re-read the contents of the i^{th} cell.

Two modules to control the access of each process to the i^{th} cell are defined. One corresponds to the behavior of the writer and the other to the behavior of the reader. The modules are shown in Figure 4.



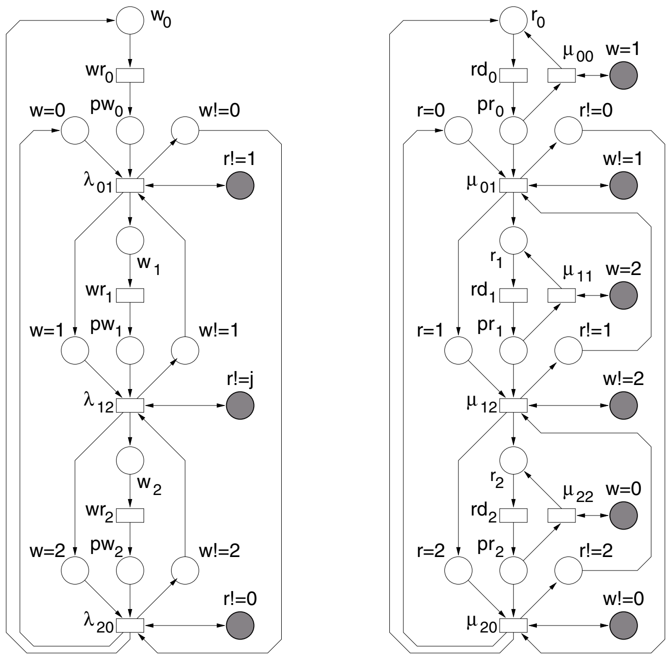
(a) writer module (b) reader module

Fig. 4. Basic modules for the writer and the reader

In Figure 4(a), a token in place w_i enables transition wr_i, that represents the action of the writer accessing the i^{th} cell. The places with label $\langle w = i \rangle$, $\langle w = j \rangle$, $\langle w \neq i \rangle$ and $\langle w \neq j \rangle$ indicate if the writer is pointing at the i^{th} or at the j^{th} cell. $\langle r \neq j \rangle$ indicates when the reader is pointing at the j^{th} cell. If transition λ_{ij} is enabled, then the reader is not pointing at cell j, the writer has just finished accessing the i^{th} cell and it can advance to the next one. The places $\langle w = i \rangle$, $\langle w = j \rangle$, $\langle w \neq i \rangle$ and $\langle w \neq j \rangle$ model the writer's control variables, and they are also used by the reader to control its own behavior. Note that $j = (i + 1) \bmod n$.

The same reasoning used to describe the writer's module also applies to the reader's. The difference is that the reader should decide to advance to the next cell or to re-read the current cell. This is captured by the two transitions in conflict, μ_{ii} and μ_{ij}. Here the decision is based on the current status of the writer, i.e. if the writer is on the j^{th} cell or not, captured by a token on places $\langle w = j \rangle$ or $\langle w \neq j \rangle$ respectively. It is easy to realize that there is a place invariant involving those places, since the sum of tokens is always equal to one, and only one of the transitions in conflict can be enabled at a time.

In order to create a process, it is only necessary to instantiate a number of modules, one for each cell, and connect them. Instantiating modules only requires replacing the i and j string by the correct cell numbers. For example, to instantiate the writer's module to control the access to the 0^{th} cell, the string i is replaced by 0 and j by 1. Connecting the modules requires to merge all the places with the same label. Figure 5 depicts the resulting Petri net models for the writer and reader of a 3-cell RRBB ACM.



(a) writer process (b) reader process

Fig. 5. The write and read processes for a 3-cell RRBB ACM

After creating the processes, they can be connected by also merging places with same label on both sides. In this case, the shadowed places in each module will be connected to some place on the other module.

Definition 2 formally introduces the concept of a module. In this definition, it is possible to see that a module is an ordinary Petri net model that has some "special" places called ports. A port is a place that models a control variable. The *local ports* model the control variables that are updated by the process to which it belongs, while the *external ports* model the control variables updated by the other process. Ports are used to identify control variables when synthesizing the source code for an ACM.

Definition 2 (Petri net module). *A Petri net module is a tuple* $MODULE = (PN, LOC, EXT, T_a, T_c)$ *such that:*

1. PN *is a Petri net structure* (P, T, F) *with:*
   (a) P *being finite set of places.*
   (b) T *being finite set of transitions.*
   (c) $F \subseteq (P \times T) \bigcup (T \times P)$ *being a set of arcs (flow relation).*
2. $LOC \subset P$ *is a finite set of* local ports.
3. $EXT \subset P$ *is a finite set of* external ports *such that* $p \in EXT \iff p\bullet = \bullet p$*. Places in* EXT *are said to be read-only.*
4. $T_a \subset T$ *is a finite set of transitions such that* $t \in T_a \iff t$ *models a media access action.*
5. $T_c \subset T$ *is a finite set of transitions such that* $t \in T_c \iff t$ *models a control action.*
6. $T_a \bigcup T_c = T$ *and* $T_a \bigcap T_c = \emptyset$*.*
7. $M_a \subset (T_a \times \mathbb{N})$ *is a relation that maps each access transition* $t \in T_a$ *into an integer that is the number of the cell addressed by* t*.*
8. $M_c \subset (T_c \times \mathbb{N} \times \mathbb{N})$ *is a relation that maps each control transition* $t \in T_c$ *into a pair of integers modeling the current and the next cells pointed by the module.*

Definitions 3 and 4 formally introduce the writer and reader basic modules, respectively.

Definition 3 (RRBB writer module). *The RRBB writer module is a tuple* $WRITER = (PN_w, LOC_w, EXT_w)$ *where:*

1. PN_w *is as defined by Figure 4(a)*
2. $LOC_w = \{\langle w = i\rangle, \langle w = j\rangle, \langle w \neq i\rangle, \langle w \neq j\rangle\}$
3. $EXT_w = \{\langle r \neq j\rangle\}$
4. $T_a = \{wri\}$
5. $T_c = \{\lambda_{ij}\}$
6. $M_a = \{(wri, i)\}$
7. $M_c = \{(\lambda_{ij}, i, j)\}$

Definition 4 (RRBB reader module). *The RRBB reader module is a tuple* $READER = (PN_r, LOC_r, EXT_r)$ *where:*

1. PN_r *is as defined by Figure 4(b)*
2. $LOC_r = \{\langle r = i\rangle, \langle r = j\rangle, \langle r \neq i\rangle, \langle r \neq j\rangle\}$
3. $EXT_r = \{\langle w = j\rangle, \langle w \neq j\rangle\}$
4. $T_a = \{rdi\}$
5. $T_c = \{\mu_{ii}, \mu_{ij}\}$
6. $M_a = \{(rdi, i)\}$
7. $M_c = \{(\mu_{ii}, i, i), (\mu_{ij}, i, j)\}$

The connection of two modules, MOD_1 and MOD_2, is defined as another Petri net module that is constructed by the union of them. Definition 5 captures this.

Definition 5 (Connection for Petri net modules). *Given two Petri net modules MOD_1 and MOD_2, where:*

- $MOD_1 = (PN_1, LOC_1, EXT_1, T_{a_1}, T_{c_1}, M_{a_1}, M_{c_1})$ *and*
- $MOD_2 = (PN_2, LOC_2, EXT_2, T_{a_2}, T_{c_2}, M_{a_2}, M_{c_2})$.

The union of them is a Petri net module $m = (PN, LOC, EXT, T_a, T_c, M_a, M_c)$ such that:

1. $PN = PN_1 \bigcup PN_2$ *where* $P = P_1 \bigcup P_2$, *If two places have the same label them they are the same,* $T = T_1 \bigcup T_2$ *and* $F = F_1 \bigcup F_2$.
2. $LOC = LOC_1 \bigcup LOC_2$.
3. $EXT = EXT_1 \bigcup EXT_2$.
4. $T_a = T_{a_1} \bigcup T_{a_2}$.
5. $T_c = T_{c_1} \bigcup T_{c_2}$.
6. $M_a = M_{a_1} \bigcup M_{a_2}$.
7. $M_c = M_{c_1} \bigcup M_{c_2}$.

The complete ACM model can also be generated by the union of the Petri net models of each resulting process. The procedure is as introduced by Definition 5 except that rules 2 and 3 do not apply.

The last required step is to set an appropriated initial marking for the Petri net model. This can be done using Definition 6.

Definition 6 (Initial marking for RRBB ACMs). *For any Petri net model of an RRBB ACM, its initial marking is defined as follows. All the places are unmarked, except in these cases:*

1. $M_0(w_i) = 1$, *if* $i = 1$.
2. $M_0(\langle w = i \rangle) = 1$, *if* $i = 1$.
3. $M_0(\langle w \neq i \rangle) = 1$, *if* $i \neq 1$.
4. $M_0(r_i) = 1$, *if* $i = 0$.
5. $M_0(\langle r = i \rangle) = 1$, *if* $i = 0$.
6. $M_0(\langle r \neq i \rangle) = 1$, *if* $i \neq 0$.

Observe that according to Definition 6, the writer is pointing at the 1^{st} cell of the ACM and reader is pointing to the 0^{th} cell. By this, it can be deduced that the ACM is assumed to be initialized with some data on its 0^{th} cell.

4.2 Verification of the Implementation Model

The Petri net model generated using the procedure discussed above will be used to synthesize source code (C++, Java, Verilog, etc.) that implements the behavior specified by the model. So, it is necessary to guarantee that such a model is correct with respect to the behavior given by Definition 1 in Section 3. In this work, it is done by applying refinement verification. In other words it is necessary to verify if the low-level specification, given by the Petri net model obtained as described above, implements correctly the abstract specification given by Definition 1.

Since Definition 1 was specified with the SMV language, it was necessary to translate the Petri net ACM model into SMV. The PEP tool [3] provides a way for translating a Petri net model into SMV and was used in our synthesis framework for such purpose.

The Petri net model specifies the mechanisms to control access to the ACM, but it does not model the data transfers. Since the goal is to check if the implementation model refines the abstract model, it is necessary to model data transfers in the implementation model. For that reason, a data array with the size of the ACM was added to the implementation model. For each event modeling a data access action, it was necessary to add the actions simulating the storage and retrieval of data in the array.

The following steps summarize what should be done to add the glue between the implementation and the abstract models.

1. Add a data array, with the same size as the ACM, to the SMV code of the Petri net model.
2. Identify in the SMV code generated by PEP the piece of code modeling the occurrence of each transition t of the Petri net model.
3. If t is a reader's action and $t \in T_a$, then the data stored in the i^{th}, where $(t, i) \in M_a$, position of the data array created in step 1 should be read.
4. If t is a writer's action and $t \in T_a$, then a new data item should be stored in the i^{th}, where $(t, i) \in M_a$, position of the data array created in step 1.

Note that the only control actions included in the model are required to avoid the non-determinism in the extra control variables. For instance, it is not desirable to allow non-deterministic changes in the values stored in the data array. By doing the above modifications in the SMV code of the generated Petri net model, it is possible to verify if the implementation model is a refinement of the abstract model with respect to the data read from the data array. It is important to note that the CTL formulae are defined in terms of the data array. Thus, if both models always read the same data from the array, and if the abstract model satisfies coherence and freshness, then the implementation model will also satisfy those properties and it can be used to synthesize the source code for the ACM.

Following the procedure described above a tool to automatically generate ACMs was designed and implemented[1]. A number of RRBBs with different sizes (starting from 3) where generated and proved to be correct for all cases.

[1] See `http://acmgen.sourceforge.net/` for details.

5 Synthesizing the Source Code

The implementation is generated from the Petri net model of each process. And the resulting source code is based on the simulation of the net model. So, the synthesis method consists of:

1. Create the shared memory as an array of the size of the desired ACM.
2. For each place p of the model, declare a Boolean variable vp named with the label of p and initialize it with the value of its initial marking. Note that if $p \in EXT$ then it will in practice be initialized by the other process, since in this case vp is seen as an external variable that belongs to another process.
3. For each transition t of the model, map into an **if** statement that is evaluated to true when all input variables of t are true. The body of the statement consists of switching the value of the input places of t to **false** and output places to **true**. If t models an access action, also add to the body of the **if** actions to write (or read) a new data item to (or from) the shared memory.

In order to perform the steps above, templates are used to define a basis for the source code of the ACM, then some gaps are fulfilled. More precisely, such gaps consist of: the declaration of the shared memory of a given size, the declarations of the control variable and the synthesis of the code that controls the access to the ACM.

Observe that the generation of the source code is performed from the Petri net model of each process and not from the model of the composed system. Algorithm 2 defines the basic procedure for the declaration and initialization of the control variables.

Algorithm 2. Control variables declaration and initialization

```
1: for all $p \in P$ do
2:    if $p \in LOC$ then
3:       Declare $p$ as a local Boolean variable
4:       Initialize variable $p$ with $M_0(p)$
5:       Make variable $p$ a shared one
6:    else if $p \in EXT$ then
7:       Create a reference to a Boolean variable $p$ that has been shared by the other
         process
8:    else
9:       Declare $p$ as a local Boolean variable
10:      Initialize variable $p$ with $M_0(p)$
11:   end if
12: end for
```

In the first case, p is declared as a local Boolean variable that can be shared with the other processes and initialized with the initial marking of p. In the second case p is a shared Boolean variable that was declared in the other process and in that case it cannot be initialized since it is a read-only control variable, from the point of view of the process being synthesized. Finally, in the third case, p is declared as a private Boolean variable and is initialized with the initial marking of p. In other words, each place will be implemented as a single bit unidirectional control variable. And each variable can be read by both processes but updated only by one of them.

Up to now the control part has not been synthesized, and there is no indication on how the data is passed from one side to the other. The shared memory can be declared statically as a shared memory segment and the only action needed to create it is to set the amount of memory that should be allocated to it.

Finally, the synthesis of the control for the reader and writer processes are introduced by Algorithms 3 and 4 respectively.

Algorithm 3. Synthesis of control for the reader

```
1: for all $t \in T$ do
2:   if $t \in T_a$ with $(t, i) \in M_a$ then
3:     Create new if statement
4:     $\forall p \in \bullet t$ add to the if condition $p = true$
5:     $\forall p \in \bullet t$ add to the if body $p = false$
6:     $\forall p \in t\bullet$ add to the if body $p = true$
7:     Add to the if body an instruction to read data from the $i^{th}$ ACM cell
8:   else if $t \in T_c$ with $(t, i, j) \in M_c$ then
9:     Create new if statement
10:    $\forall p \in \bullet t$ add to the if condition $p = true$
11:    $\forall p \in \bullet t$ add to the if body $p = false$
12:    $\forall p \in t\bullet$ add to the if body $p = true$
13:  end if
14: end for
```

In Algorithm 3, the first case captures the synthesis of control to a data *read transition* addressing the i^{th} cell. The condition to the control is given by the pre-set of t and if it is satisfied then its pre-set it switched to $false$ and its post-set to $true$. And some data is read from the i^{th} cell. The second captures the synthesis of control to a *control transition*. As in the previous the condition is given by the pre-set of t and then its pre-set it switched to $false$ and its post-set to $true$.

Algorithm 4 is similar to Algorithm 3. The difference is that instead of reading some data from the i^{th} cell, the process will write some data into it.

The approach described here was used in the generation of C++ implementations for ACMs. In Figures 6 and 7 the methods that perform the shared memory accesses and control actions to the 3-cell RRBB ACM introduced in Section 4 are shown.

In Figure 6(a) it is possible to see the method that actually writes some data into the ACM. Line 3 captures the transition wr_0 in the Petri net model enabled. In this case: the variables implementing its pre-set are turned to `false`, line 4; the variables implementing its post-set are turned to `true`, line 5; and some

Algorithm 4. Synthesis of control for the writer

```
 1: for all $t \in T$ do
 2:   if $t \in T_a$ with $(t, i) \in M_a$ then
 3:     Create new if statement
 4:     $\forall p \in \bullet t$ add to the if condition $p = true$
 5:     $\forall p \in \bullet t$ add to the if body $p = false$
 6:     $\forall p \in t\bullet$ add to the if body $p = true$
 7:     Add to the if body a instruction to write new data on the $i^{th}$ ACM cell
 8:   else if $t \in T_c$ with $(t, i, j) \in M_c$ then
 9:     Create new if statement
10:     $\forall p \in \bullet t$ add to the if condition $p = true$
11:     $\forall p \in \bullet t$ add to the if body $p = false$
12:     $\forall p \in t\bullet$ add to the if body $p = true$
13:   end if
14: end for
```

data is written into the 0^{th} cell of the ACM, line 6. Note that the `val` is the new data to be sent and `shm_data` implements the shared memory. Note that each **if** statement refers to some transition in the Petri net model.

```
1.  void Writer::Send(acm_t val) {
2.
3.    if (w0 == true) { //wr0
4.      w0 = false;
5.      pw0 = true;
6.      *(shm_data + 0) = val;
7.    } else if (w1 == true) { //wr1
8.      w1 = false;
9.      pw1 = true;
10.     *(shm_data + 1) = val;
11.   } else if (w2 == true) { //wr2
12.     w2 = false;
13.     pw2 = true;
14.     *(shm_data + 2) = val;
15.   }
16. }
```

(a) Writer::Send()

```
1.  acm_t Reader::Receive(void) {
2.
3.    acm_t val;
4.
5.    if (r0 == true) {
6.      r0 = false;
7.      pr0 = true;
8.      val = *(shm_data + 0);
9.    } else if (r1 == true) {
10.     r1 = false;
11.     pr1 = true;
12.     val = *(shm_data + 1);
13.   } else if (r2 == true) {
14.     r2 = false;
15.     pr2 = true;
16.     val = *(shm_data + 2);
17.   }
18.
19.   return(val);
20. }
```

(b) Reader::Receive()

Fig. 6. Access actions implementation

The same reasoning applies to the reader access method shown in Figure 6(b). The only difference is that instead of writing into the ACM, it reads from there.

The methods implementing the control actions are somewhat more complex, but follow the same principle. The implementation of the writer's control actions are given by the method in Figure 7(a). As before, the same idea is used, implementing each control transition as an **if** statement whose condition is given by the variables of the pre-set and the body consists of switching the pre-set to `false` and the post-set to `true`. For example, the code implementing the firing of transition λ_{01} is given by lines 3 to 14 of Figure 7(a). Observe that `we0` and `wne0` stands for $w = 0$ and $w \neq 0$ respectively.

The writer's control actions are inside an infinite loop whose last instruction is a call to a `pause()`[2] function. This is done because if there is no λ transition enabled, with the writer pointing at the i^{th} cell, it means that the reader is pointing at the $(i+1)^{th}$ cell. And in this case the writer should wait for the reader to execute. By using the `pause()` function in line 17, busy waiting algorithms are avoided. Also, note that the exit from the loop is done by a `break` statement, as in line 13.

```
1.  void Writer::Lambda(void) {
2.    while (true) {
3.      if (*we0 == true &&
4.          *wne1 == true &&
5.          w0p == true &&
6.          *rne1 == true) { // l0_1
7.        *we0 = false;
8.        *wne1 = false;
9.        w0p = false;
10.       w1 = true;
11.       *wne0 = true;
12.       *we1 = true;
13.       break;
14.     }
15.     if (...) {...} // l1_2
16.     if (...) {...} // l2_0
17.     pause();
18.   }
19. }
```

(a) Writer::Lambda()

```
1.  void Reader::Mu(void) {
2.    if (r0p == true &&
3.        *we1 == true) { // m0_0
4.      r0p = false;
5.      r0 = true;
6.      kill(pair_pid, SIGCONT);
7.    } else if (*re0 == true &&
8.        *rne1 == true &&
9.        r0p == true &&
10.       *wne1 == true) { // m0_1
11.     *re0 = false;
12.     *rne1 = false;
13.     r0p = false;
14.     r1 = true;
15.     *rne0 = true;
16.     *re1 = true;
17.     kill(pair_pid, SIGCONT);
18.   } else if (...) {... // m1_1
19.   } else if (...) {... // m1_2
20.   } else if (...) {... // m2_2
21.   } else if (...) {...} // m2_0
22. }
```

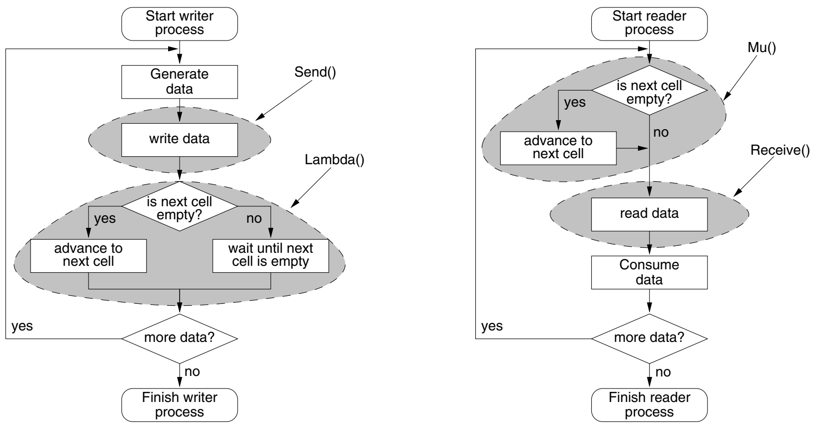
(b) Reader::Mu()

Fig. 7. Control actions implementation

The control actions of the reader process are implemented by the method in Figure 7(b). Again, each transition is implemented as an **if** statement. For instance, μ_{00} is implement by the code from line 2 to 6 and μ_{01} is implemented in lines 7 to 17. It is important to observe that every time the reader executes a control actions, it sends the signal `SIGCONT` to the writer, as in lines 6 and 17. This is to wake up the writer in the case it is sleeping due to a `pause()`.

Finally, the methods generated above need to be integrated into the communicating processes. As explained before and shown in Figure 8, the writer first calls the Send() and then the Lambda() methods. On the other hand, the reader first calls the Mu() and then the Receive() methods. In the code generated these operations are encapsulated into two public methods: Write() and Read(), available for the writer and reader respectively. With this, the correct use of the communication scheme is ensured.

[2] The pause() library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.



(a) Writer flow (b) Reader flow

Fig. 8. Flowchart for communicating processes

In this Section an automatic approach to generate source code from Petri net models was discussed. The algorithms introduced here only gives conceptual ideas on what needs to be done for the synthesis of the code. When executing the procedure, many details related to the target programming language has to be taken into account. The algorithms above were implemented to generate C++ code to be executed on a Linux system. The reader should consult [7] for more details on creating shared memory segments on UNIX systems.

6 Conclusions and Future Work

This work introduces a novel approach to the automatic synthesis of ACMs. The method presented here is based on the use of modules to the generation of Petri net models that can be verified against a more abstract specification.

Firstly, the behavior of RRBB ACMs was formally defined and the properties it should satisfy were described by CTL formulas. Then the procedure of generating the Petri net models was presented, including the definition of the basic modules and the algorithms required to instantiate and connect them. It was argued how the the resulting model is translated to an SMV model in order to be verified against the more abstract model defined in the beginning of the process. Finally, a C++ implementation is generated from the Petri net model.

Compared to the previous work [1], the method of generating Petri net models introduced here has the disadvantage of requiring model checking to guarantee its correctness. In the previous approach based on ACM regions, it was guaranteed by construction. However, the cost of executing the ACM regions algorithms is too high. And when it becomes limited by the state-space explosion problem, no implementation Petri net model could be generated and synthesis fails. In the approach proposed here, state-space explosion is limited to the verification of the

Petri net implementation model. This step is off the design flow (see Figure 3). Thus we could generate C++ codes from the implementation model whether it can be verified or not. An unverified implementation nonetheless has practical engineering significances because the Petri net model is highly regular and its behavior can be inferred from that of similar ACMs of smaller and verifiable size.

The next step into the direction of the automatic generation of ACMs is to provide a formal proof that the procedure of generating the net models is correct by design. With this, it will be possible to skip the verification step. And the time required to synthesize devices that can be trusted will drastically reduce. Also it is necessary to introduce formally the mechanisms used in the overwriting ACM classes. Finally, it is a primary goal to be able to generate the ACMs in the form of a Verilog code that can be used to synthesize a piece of hardware.

Acknowledgments. This work has been supported by CICYT TIN2004-07925, a Distinction for Research from the Generalitat de Catalunya and by the EPSRC (grant EP/C512812/1) at the University of Newcastle upon Tyne.

References

1. Cortadella, J., Gorgônio, K., Xia, F., Yakovlev, A.: Automating synthesis of asynchronous communication mechanisms. In: Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05), St. Malo, France, June 2005, pp. 166–175. IEEE Computer Society, Washington (2005)
2. Fassino, J-P.: THINK: vers une architecture de systèmes flexibles. PhD thesis, École Nationale Supérieure des Télécommunications (December 2001)
3. Grahlmann, B.: The pep tool. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 440–443. Springer, Heidelberg (1997)
4. Lamport, L.: On interprocess communication — parts I and II. Distributed Computing 1(2), 77–101 (1986)
5. McMillan, K.L.: The SMV System: for SMV version 2.5.4 (November 2000) Available from `http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps.gz`
6. Simpson, H.R.: Protocols for process interaction. IEE Proceedings on Computers and Digital Techniques 150(3), 157–182 (May 2003)
7. Richard Stevens, W.: Advanced programming in the UNIX environment. Addison Wesley Longman Publishing Co., Inc, Redwood City, CA, USA (1992)
8. Xia, F., Hao, F., Clark, I., Yakovlev, A., Chester, G.: Buffered asynchronous communication mechanisms. In: Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04), pp. 36–44. IEEE Computer Society, Washington (2004)
9. Yakovlev, A., Kinniment, D. J., Xia, F., Koelmans, A. M.: A fifo buffer with non-blocking interface. TCVLSI Technical Bulletin, pp. 11–14, Fall (1998)
10. Yakovlev, A., Xia, F.: Towards synthesis of asynchronous communication algorithms. In: Caillaud, B., Darondean, P., Lavagno, L., Xie, X. (eds.) Synthesis and Control of Discrete Event Systems. Part I: Decentralized Systems and Control, pp. 53–75. Kluwer Academic Publishers, Boston (January 2002)

History-Dependent Petri Nets

Kees van Hee, Alexander Serebrenik, Natalia Sidorova, and Wil van der Aalst

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{k.m.v.hee,a.serebrenik,n.sidorova,w.m.p.v.d.aalst}@tue.nl

Abstract. Most information systems that are driven by process models (e.g., workflow management systems) record events in event logs, also known as transaction logs or audit trails. We consider processes that not only keep track of their history in a log, but also make decisions based on this log. To model such processes we extend the basic Petri net framework with the notion of history and add guards to transitions evaluated on the process history. We show that some classes of history-dependent nets can be automatically converted to classical Petri nets for analysis purposes. These classes are characterized by the form of the guards (e.g., LTL guards) and sometimes the additional requirement that the underlying classical Petri net is either bounded or has finite synchronization distances.

1 Introduction

Numerous state-of-the-art enterprise information systems contain a workflow engine, which keeps track of all events as a part of its basic functionality. In this paper we consider processes that not only record the events but also make choices based on the previous events, i.e. based on their *history*. The ability of a system to change its behavior depending on its observed behavior is known as *adaptivity* and in this sense this paper is about a special class of adaptive systems.

In classical Petri nets the enabling of a transition depends only on the availability of tokens in the input places of the transition. We extend the model by recording the history of the process and introducing transition guards evaluated on the history. To illustrate the use of history, we consider a simple example of two traffic lights on crossing roads.

Example 1. Figure 1 (left) presents two traffic lights, each modelled by a cycle of three places and three transitions. The places model the states of each traffic light (red, green and yellow), and the transitions change the lights from one color to the next color. We assume that in the initial state both lights are red.

We want the system to be safe and fair, i.e., the traffic lights are never green at the same time, the right traffic light can become green at most R times more than the left traffic light, and similarly, the left traffic light can become green at most L times more than the right traffic light. Usually one takes $R = 1$ and

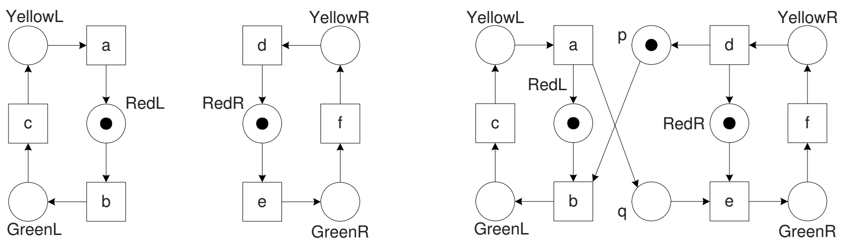


Fig. 1. Traffic lights: without restrictions (*left*) and alternating (*right*)

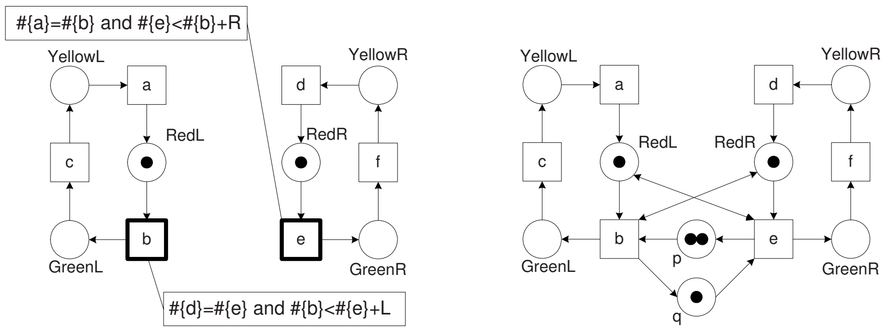


Fig. 2. A history-dependent Petri net with parameters R and L (*left*) and the history guards replaced according to Theorem 23 for $R = 1$ and $L = 2$ (*right*)

$L = 0$, or $R = 0$ and $L = 1$, implying alternating behavior of the traffic lights. In order to obtain the alternating behavior one traditionally adds control places p and q as in the right-hand side of Figure 1. This figure models the situation with $R = 0$ and $L = 1$. Note that it is not easy to generalize this construction for arbitrary R and L.

Our approach consists in making the guards explicit as shown in left-hand side of Figure 2. To ensure safety, we require that b can fire only if the right traffic light is red, i.e., transitions d and e have fired the same number of times. The guard of b is written then as $\#\{d\} = \#\{e\}$. Similarly, e obtains the guard $\#\{a\} = \#\{b\}$. In order to guarantee fairness, we require that in any history, b fires at most L times more than e, i.e. $\#\{b\} \leq \#\{e\} + L$, and e fires at most R times more than b, i.e., $\#\{e\} \leq \#\{b\} + R$. To ensure this we add the additional requirement $\#\{b\} < \#\{e\} + L$ to the guard of b and the additional requirement $\#\{e\} < \#\{b\} + R$ to the guard of e. This results in the history-dependent Petri net shown in Figure 2 (left).

Using history we can separate the modeling of the standard process information (switching the traffic light to the following color) from additional requirements ensuring the desired behavior. Hence, we believe that introducing history-dependent guards amounts to enhanced modeling comfort. Observe also that global access to the history allows to ease modeling of synchronous choices. Assume that at a certain point a choice has to be made between transitions a and b. Assume further that the only impact of this choice is somewhere later in the process: a' has to be chosen if a has been chosen and b' has to be chosen if b has been chosen. A classical solution of this problem involves creating two places p_a and p_b with the only incoming arc coming from a (b) and the only outgoing arc leading to a' (b'). Rather than cluttering our model with additional places, we set the guard of a' (b') to demand that a (b) has been chosen before.

In this paper we consider two approaches to introduce history into the Petri net model: (1) *token history*, where each individual token carries its own history, i.e., history can be seen as special kind of color, and (2) *global history*, where there is a single centralized history and every transition guard is evaluated on it (like in our traffic lights example). Token history can be used in distributed settings where different components do not have information about the actions of other components. Global history is in fact a special case of token history for transparent systems where all components are aware of the actions of other components.

By introducing history-dependent guards, we increase the expressive power. On the traffic lights example, we can easily see that we can check the emptiness of a place using history: $RedR$ is empty if and only if $\#\{e\} - \#\{d\} = 1$. Hence, we can model inhibitor arcs and consequently our formalism is Turing complete. Since, we are interested not only in modeling but also in verification, we identify a number of important classes of global history nets (e.g. nets with LTL guards) that can be transformed to bisimilar classical Petri nets and provide corresponding transformations. For instance, the history-dependent net on the left-hand side of Figure 2 can be automatically transformed to the classical net on the right-hand side (we took $R = 1$ and $L = 2$).

Due to the Turing completeness, not every history-dependent net can be represented by a classical Petri net. We are still interested in simulation and validation of history-dependent nets. Simulation and validation are however complicated by the fact that the representation of the current state of the system requires in general an unbounded amount of memory, due to the growth of the history. We solve this problem for a Turing complete subclass of global history nets (in which we use event counting, but not event precedence in the guards) by defining a transformation to bisimilar inhibitor nets. Inhibitor nets, though being Turing complete, have a state representation of a fixed length (a marking), which makes the simulation and validation feasible.

The remainder of the paper is organized as follows. After some preliminary remarks in Section 2, we introduce the notion of *event history* together with a *history logic* in Section 3. Section 4 introduces *token history nets* and Section 5 introduces *global history nets*. In Section 6 we show how to map several subclasses of global history nets with counting formulae as guards to classical Petri nets or inhibitor Petri nets, and in Section 7 we describe a transformation of global history nets with LTL guards to classical Petri nets. Finally, we review the related work and conclude the paper.

2 Preliminaries

$\mathbb{N}$ denotes the set of natural numbers and $\mathbb{Z}$ the set of integers.

Let P be a set. A *bag (multiset)* m over P is a mapping $m : P \to \mathbb{N}$. We identify a bag with all elements occurring only once with the set containing the elements of the bag. The set of all bags over P is denoted by $\mathbb{N}^P$. We use $+$ and $-$ for the sum and the difference of two bags and $=, <, >, \leq$ and $\geq$ for the comparison of bags, which are defined in a standard way. We overload the set notation, writing $\emptyset$ for the empty bag and $\in$ for the element inclusion. We write e.g. $m = 2[p] + [q]$ for a bag m with $m(p) = 2$, $m(q) = 1$, and $m(x) = 0$ for all $x \notin \{p, q\}$. As usual, $|m|$ and $|S|$ stand for the number of elements in bag m and in set S, respectively.

For (finite) *sequences* of elements over a set P we use the following notation: The empty sequence is denoted with ϵ; a non-empty sequence can be given by listing its elements.

A *transition system* is a tuple $E = \langle S, Act, T\rangle$ where S is a set of *states*, Act is a finite set of *action names* and $T \subseteq S \times Act \times S$ is a *transition relation*. We say that E is finite if S is finite. A *process* is a pair (E, s_0) where E is a transition system and $s_0 \in S$ an initial state. We denote $(s_1, a, s_2) \in T$ as $s_1 \xrightarrow{a}_E s_2$, and we say that a leads from s_1 to s_2 in E. We omit E and write $s \xrightarrow{a} s'$ whenever no ambiguity can arise. For a sequence of action names $\sigma = a_1 \ldots a_n$ we write $s_1 \xrightarrow{\sigma} s_2$ when $s_1 = s^0 \xrightarrow{a_1} s^1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s^n = s_2$. Next, $s_1 \xrightarrow{*} s_2$ means that there exists a sequence $\sigma \in T^*$ such that $s_1 \xrightarrow{\sigma} s_2$. We say that s_2 is *reachable* from s_1 if and only if $s_1 \xrightarrow{*} s_2$. Finally, the language of a process (E, s_0), denoted $\mathfrak{L}(E, s_0)$, is defined as $\{\sigma \mid \sigma \in T^*, \exists s : s_0 \xrightarrow{\sigma} s\}$.

Definition 2. *Let $E_1 = \langle S_1, Act, T_1\rangle, E_2 = \langle S_2, Act, T_2\rangle$ be transition systems. A relation $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a* simulation *if and only if for all $s_1, s_1' \in \mathcal{S}_1$, $s_2 \in \mathcal{S}_2$, $s_1 \xrightarrow{a}_{E_1} s_1'$ implies that $s_2 \xrightarrow{a}_{E_2} s_2'$ and $s_1'\ R\ s_2'$ for some $s_2' \in \mathcal{S}_2$.*

E_1 and E_2 are bisimilar *if there exists a relation $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ such that both R and R^{-1} are simulations.*

Next we introduce a number of notions related to Petri nets.

Definition 3. *A* Petri net *N over a fixed set of labels Σ is a tuple $\langle P, T, F, \Lambda\rangle$, where: (1) P and T are two disjoint non-empty finite sets of* places *and* transitions *respectively; we call the elements of the set $P \cup T$* nodes *of N; (2) $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a* flow relation *mapping pairs of places and transitions to the naturals; (3) $\Lambda : T \to \Sigma$ is a* labeling function *that maps transitions of T to action labels from Σ.*

An inhibitor net *is a tuple $\langle P, T, F, \Lambda, I\rangle$ such that $\langle P, T, F, \Lambda\rangle$ is a Petri net and $I \subseteq P \times T$ is a set of* inhibitor arcs.

We present nets with the usual graphical notation. For any pair of nodes x, y with $F(x, y) \geq 1$, we say that (x, y) is an arc with *weight* $F(x, y)$.

Given a transition $t \in T$, the *preset* $^\bullet t$ and the *postset* $t^\bullet$ of t are the *bags* of places where every $p \in P$ occurs $F(p,t)$ times in $^\bullet t$ and $F(t,p)$ times in $t^\bullet$. Analogously we write $^\bullet p, p^\bullet$ for pre- and postsets of places.

A marking m of N is a bag over P; markings are states (configurations) of a net. A pair (N, m) is called a *marked* Petri net. A transition $t \in T$ is *enabled* in marking m if and only if $^\bullet t \leq m$ and moreover, for inhibitor nets, $m(p) = 0$ for any p such that $(p,t) \in I$. An enabled transition t may *fire*. This results in a new marking m' defined by $m' \overset{\text{def}}{=} m - {}^\bullet t + t^\bullet$. We interpret a labeled Petri net N as a transition system/process $\langle \mathbb{N}^P, \Lambda(T), \longrightarrow\rangle$ / $(\langle \mathbb{N}^P, \Lambda(T), \longrightarrow\rangle, m_0)$ respectively, where markings play the role of states and labels of the firing transitions play the role of action names. The notion of reachability for Petri nets is inherited from the transition systems. We denote the set of all markings reachable in net N from marking m as $\mathcal{R}_N(m)$. We will drop N and write $\mathcal{R}(m)$ when no ambiguity can arise. A marked net (N, m_0) is called *bounded* if its reachability set is finite.

3 Event History and History Logic

In this section we present the general notion of event history. In the coming sections we investigate two kinds of nets that use event history: *token history nets* and *global history nets.*

One might expect an event history to be a totally ordered series of events. However, information on the relative order of events registered by different components might be missing. Therefore, we define a history as a partial order.

Definition 4. *Given a set of action labels Σ, a* history *is a labeled poset, i.e., a triple $\langle E, \prec, \lambda\rangle$, where E is a set of* events *coming from a fixed universe, $\prec$ is a partial order on E and $\lambda : E \to \Sigma$ is a* labeling function. *If $E = \emptyset$ the corresponding history is called* the empty history *and denoted by ϵ.*

Two histories $\langle E_1, \prec_1, \lambda_1\rangle$ and $\langle E_2, \prec_2, \lambda_2\rangle$ are consistent *if and only if the transitive closure of $\prec_1 \cup \prec_2$ is a partial order for $E_1 \cup E_2$ and $\lambda_1(e)$ coincides with $\lambda_2(e)$ for any $e \in E_1 \cap E_2$.*

We define two operations to create a new history out of existing histories: *extension and union.*

Definition 5. *The* extension $\langle E, \prec, \lambda\rangle :: \ell$ *of a history $\langle E, \prec, \lambda\rangle$ with an event labeled by ℓ is the history $\langle E \cup \{e\}, \prec_\ell, \lambda_\ell\rangle$, where e is a new event,*[1] *$\prec_\ell$ is defined as $\prec \cup \{(x, e) \mid x \in E\}$ and λ_ℓ maps e to ℓ and coincides with λ on E.*

The union $\langle E_1, \prec_1, \lambda_1\rangle \cup \langle E_2, \prec_2, \lambda_2\rangle$ of consistent histories *is defined as $\langle E_1 \cup E_2, \prec, \lambda_1 \cup \lambda_2\rangle$, where $\prec$ is the transitive closure of $\prec_1 \cup \prec_2$.*

These operations will be used in the next sections on token history and global history for Petri nets. In global history nets each firing of a transition extends

[1] Note that it is essential that e is a "fresh" identifier not present in E but also not used in any "known" history.

the global history. In token history nets, tokens created by a transition firing carry the union of histories of the consumed tokens extended with the firing event.

Next we present a language of history-dependent predicates that will be used in the guards of history-dependent nets. From here on we assume a countable set Var of variables to be given.

Definition 6. *Given a set Σ of labels and $x \in \mathit{Var}$, we define a formula φ, a term q and a label expression l over Σ as follows:*

$$\begin{array}{lll} \varphi ::= \mathit{false} \mid \varphi \Rightarrow \varphi & \mid x \preceq x & \mid q < q \mid l == l \\ q ::= \mathbb{N} & \mid (\#\mathit{Var} : \varphi) \mid (q+q) & \\ l ::= \Sigma & \mid \lambda(x) & \end{array}$$

Sets of formulae, terms and label expressions over Σ are denoted as $\mathcal{F}_\Sigma$, $\mathcal{Q}_\Sigma$ and $\mathcal{L}_\Sigma$, respectively.

Using the definition above we can define the following short-hand notations in the standard way: *true*, $\neg$, $\wedge$, $\vee$, $>$, $\geq$, $\leq$, $=$ (comparisons of terms). We omit brackets if this does not introduces ambiguities. The counting operator $\#$ is powerful enough to express the standard quantifiers: We write $\exists x : \varphi$ for $(\#x : \varphi) > 0$ and $\forall x : \varphi$ for $(\#x : \varphi) = (\#x : \mathit{true})$. For a finite set of labels $S = \{s_1, \ldots, s_n\}$, $\ell \in S$ stands for $(\ell == s_1 \vee \ldots \vee \ell == s_n)$ and $\#S$ stands for $(\#x : \lambda(x) \in S)$. Finally $e_1 \prec e_2$ means that $(e_1 \preceq e_2) \wedge \neg(e_2 \preceq e_1)$.

In order to define the semantics we introduce the notion of an *assignment* defined as a mapping of variables from Var to events from E. Given a variable x, an event e and an assignment ν, $\nu[x \rightarrow e]$ denotes the assignment that coincides with ν for all variables except for x which is mapped to e.

Definition 7. *Given a history $H = \langle E, \prec, \lambda\rangle$ and an assignment ν, the* evaluation *eval and the truth value of a* formula *are defined by mutual structural induction. The evaluation function eval maps a* term *q to $\mathbb{N}$ as follows:*

$$\mathit{eval}(H,\nu,q) = \begin{cases} q & \text{if } q \in \mathbb{N}; \\ \left|\{e \in E \mid \langle H, \nu[x \rightarrow e]\rangle \models \varphi\}\right| & \text{if } q \text{ is } \#x : \varphi; \\ \mathit{eval}(H,\nu,q_1) + \mathit{eval}(H,\nu,q_2) & \text{if } q \text{ is } q_1 + q_2. \end{cases}$$

Similarly, eval maps a label expression *l to Σ:*

$$\mathit{eval}(H,\nu,l) = \begin{cases} l & \text{if } l \in \Sigma; \\ \lambda(\nu(x)) & \text{if } l \text{ is } \lambda(x). \end{cases}$$

Finally, the truth value of a formula *is defined as follows:*

- $\langle H,\nu\rangle \models \mathit{false}$ *is always false;*
- $\langle H,\nu\rangle \models \varphi_1 \Rightarrow \varphi_2$ *if not* $\langle H,\nu\rangle \models \varphi_1$ *or* $\langle H,\nu\rangle \models \varphi_2$*;*
- $\langle H,\nu\rangle \models x_1 \preceq x_2$ *if* $\nu(x_1) \prec \nu(x_2)$ *or* $\nu(x_1)$ *coincides with* $\nu(x_2)$*;*

– $\langle H, \nu\rangle \models q_1 < q_2$ *if* $eval(H, \nu, q_1) < eval(H, \nu, q_2)$ *($<$ is the standard order on the naturals);*
– $\langle H, \nu\rangle \models l_1 == l_2$ *if* $eval(H, \nu, l_1)$ *coincides with* $eval(H, \nu, l_2)$.

One can show that for *closed terms* and *formulae*, i.e., terms and formulae where all variables appear in the scope of #, the result of the evaluation does not depend on ν. Therefore, for a closed term q we also write $eval(H, q)$ and for a closed formula φ we also write $H \models \varphi$. The set of closed formulae over Σ is denoted $\mathcal{CF}_\Sigma$.

To illustrate our language, we return to the traffic light example from Figure 2. The guards of transitions are formulated according to Definition 6.

4 Token History Nets

In this section we introduce token history nets as a special class of colored Petri nets [11] with history as color. The tokens of an initial marking have an empty history and every firing of a transition t produces tokens carrying the union of the histories of the consumed tokens extended with the last event, namely the firing of transition t labeled by $\Lambda(t)$.

Definition 8. *A* token history net N *is a tuple* $\langle P, T, F, \Lambda, g\rangle$ *such that* $N_P = \langle P, T, F, \Lambda\rangle$ *is a labeled Petri net and* $g : T \to \mathcal{CF}_{\Lambda(T)}$ *defines the* transition guards.

The semantics of a token history net is given by the transition system defined as follows:

Color is the set of possible histories $\langle E, \prec, \lambda\rangle$ *over the label set* $\Lambda(T)$*. A* state m *of a token history net* N *is a bag of tokens with histories as token colors, i.e., a marking* $m : (P \times Color) \to \mathbb{N}$*.*

The transition relation *is specified by:* $m \xrightarrow{a} m'$ *if and only if there exist a transition* t *with* $\Lambda(t) = a$*, a history* H *and two bags* $cons$ *and* $prod$ *of tokens such that:*

– $H = \bigcup_{(p,c)\in cons} c$ *(H is the unified history),*
– $cons \leq m$ *(tokens from $cons$ are present in m),*
– $\sum_{(p,c)\in cons}[p] = {}^\bullet t$ *(tokens are consumed from the right places),*
– $prod = \sum_{p\in t^\bullet}[(p, H :: \Lambda(t))]$ *($prod$ is the bag of tokens to be produced),*
– $m' = m - cons + prod$*, and*
– $H \models g(t)$ *(i.e., the guard evaluates to true given the unified history H).*

A token history net is thus defined by attaching a guard to all transitions of a classical Petri net. A transition guard is evaluated on the union H of histories of consumed tokens. Recall that the union of two histories is defined for consistent histories only. We will call a marking *consistent* if the union of all its token histories is defined. The following lemma states that consistency of markings is an invariant property (observe that a transition firing cannot destroy consistency).

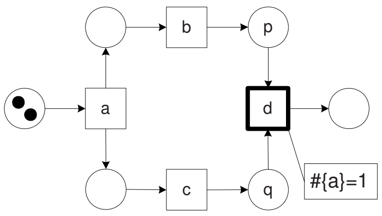


Fig. 3. A token history net

Lemma 9. *Let m be a consistent marking and $m \xrightarrow{*} m'$ for some marking m'. Then m' is consistent.*

Proof. Proof of the lemma relies on the fact that a *fresh* event is used every time histories are extended.

To conclude this section we illustrate the semantics of token history nets.

Example 10. Consider the token history net in Figure 3. Firings of transition d are allowed iff there is only one event labeled by a in the union of the histories of tokens consumed from places p and q, i.e. tokens on p and q originate from the same initial token. Let the sequence $abcabc$ fire from the initial marking, which results in the marking $m = [(p, H_1)] + [(p, H_2)] + [(q, H_3)] + [(q, H_4)]$ with $H_1 = \langle\{e_1, e_2\}, \{e_1 \prec e_2\}, \{(e_1, a), (e_2, b)\}\rangle$, $H_2 = \langle\{e_4, e_5\}, \{e_4 \prec e_5\}, \{(e_4, a), (e_5, b)\}\rangle$, $H_3 = \langle\{e_1, e_3\}, \{e_1 \prec e_3\}, \{(e_1, a), (e_3, c)\}\rangle$ and $H_4 = \langle\{e_4, e_6\}, \{e_4 \prec e_6\}, \{(e_4, a), (e_6, c)\}\rangle$. The transition labeled d can fire consuming tokens $[(p, H_1)]$ and $[(q, H_3)]$ since the tokens share event e_1 in their history. The produced token is $[(s, H_5)]$ with $H_5 = \langle\{e_1, e_2, e_3, e_7\}, \{e_1 \prec e_2, e_1 \prec e_3, e_1 \prec e_7, e_2 \prec e_7, e_3 \prec e_7\}, \{(e_1, a), (e_2, b), (e_3, c), (e_7, d)\}\rangle$. This transition cannot fire on e.g. $[(p, H_1)]$ and $[(q, H_4)]$ since the union $H_1 \cup H_4$ contains two events (e_1 and e_4) labeled by a while the guard specifies that the number of a events should be one ($\#\{a\} = 1$). Token history allows thus distinguishing between tokens originating from different firings of the same transition, i.e., mimicking another popular color, namely case identifiers.

5 Global History Nets

In this section we introduce global history nets, where history is a separate object accessible when the guards of transitions are evaluated.

Definition 11. *A* global history net *N is a tuple $\langle P, T, F, \Lambda, g\rangle$ such that $N_P = \langle P, T, F, \Lambda\rangle$ is a labeled Petri net and $g : T \to \mathcal{CF}_{\Lambda(T)}$ defines the* transition guards*.*

The semantics of global history nets is defined as follows:

A state *of N is a pair (m, H) where m is a marking of N_P and H is a history over $\Lambda(T)$. The* transition relation *is specified by: $(m, H) \xrightarrow{a} (m', H')$ if and* *only if there exists $t \in T$ such that $\lambda(t) = a$, ${}^\bullet t \leq m$, $H \models g(t)$, $m' = m - {}^\bullet t + t^\bullet$ and H' is $H :: \Lambda(t)$.*

Given a global history net N we denote by $\mathcal{S}(N)$ the set of all states of the net. Analogously to marked Petri nets we consider marked global history nets being pairs $(N, (m, H))$ such that N is a global history net and $(m, H) \in \mathcal{S}(N)$. The set of states reachable from (m, H) in N is denoted $\mathcal{R}_N((m, H))$; the set of states reachable from an initial state (m_0, ϵ) is thus $\mathcal{R}_N((m_0, \epsilon))$.

The interleaving semantics results in the following property:

Proposition 12. *Let $N = \langle P, T, F, \Lambda, g\rangle$ be a global history net and $(m, \langle E, \prec, \lambda\rangle) \in \mathcal{R}_N((m_0, \epsilon))$. Then $\prec$ is a total order on E.*

Note that history does not contain information which transitions exactly have fired, but labels of those transitions only. Therefore, knowing the initial marking and the history, we cannot reconstruct the current marking in general. However, it can easily be seen that if Λ is injective the current marking can be derived from the initial marking an history.

Proposition 13. *Let $N = \langle P, T, F, \Lambda, g\rangle$ be a global history net such that Λ is injective. Then, for a given H: $(m_1, H), (m_2, H) \in \mathcal{R}_N((m_0, \epsilon))$ implies $m_1 = m_2$.*

This proposition implies that we are able to express conditions on the marking by using global history nets with *injective labeling*. To illustrate this, we introduce $\#^\bullet p$ as a shorthand for $\sum_{t \in {}^\bullet p} \#\{\Lambda(t)\}$ for some place p, i.e., $\#^\bullet p$ is the number of tokens *produced* to the place p. Similarly, $\#p^\bullet$ denotes $\sum_{t \in p^\bullet} \#\{\Lambda(t)\}$, i.e., the number of tokens *consumed* from p according to the history. (Note that the sum is taken over a bag.) Now, let m_0 be the initial marking of a global history net N where Λ is injective, and assume $(m, H) \in \mathcal{R}_N((m_0, \epsilon))$. Clearly, $m(p) = m_0(p) - \#p^\bullet + \#^\bullet p$ for any $p \in P$. Hence, we can express any condition on the current state in a transition guard. For example, we can simulate inhibitor arcs by adding the condition $m_0(p) - \#p^\bullet + \#^\bullet p = 0$. Since inhibitor nets are known to be Turing complete (cf. [17]), global history nets with unique labels are Turing complete as well.

Corollary 14. *Global history nets $N = \langle P, T, F, \Lambda, g\rangle$ are Turing complete.*

Next we discuss the implications of Corollary 14 on the expressive power of token history nets.

Token history vs. global history. Observe that in general it is impossible to derive the corresponding token histories from the history of a global history net. Consider the net from Figure 3 as a global history net and suppose that its global history is *aabc*. One cannot derive whether the tokens on places p and q will share the history event labeled by a or not. On the other hand, in general it is impossible to reconstruct the corresponding global history from a given marking of a token history net, since no information is available on the order of truly concurrent firings. So marking m from Example 10 can be obtained as a result of firing sequences *abcabc*, *aabbcc*, *abacbc*, etc. and have the corresponding global history. We can however *mimic* a global history net with a token history net.

The key idea behind our construction is adding a new place p^* with one initial token, connected to all transitions. Since the token in p^* is updated at each firing, it will keep a *global log*. Since all transitions are connected to p^*, their guards will be evaluated on the same history as in the original global history net. Formally, given a global history net $N = \langle P, T, F, \Lambda, g\rangle$ with initial marking m_0, we construct a token history net $N' = \langle P', T, F', \Lambda, g\rangle$ with initial marking m_0' such that $P' = P \cup \{p^*\}$ (with $p^* \notin P$ being the new place), $F'(n_1, n_2) = F(n_1, n_2)$ for $(n_1, n_2) \in (P \times T) \cup (T \times P)$ and $F'(n_1, n_2) = 1$ for $(n_1, n_2) \in (\{p^*\} \times T) \cup (T \times \{p^*\})$, and $\forall p \in P : m_0'((p, \epsilon)) = m_0(p)$, $m_0'((p^*, \epsilon)) = 1$ and $m_0'(x) = 0$ in all other cases. N' is called the *log extension* of N. It is easy to show that both nets are indeed bisimilar.

Lemma 15. *(N, m_0) and (N', m_0') as above are bisimilar.*

Proof. (Idea) Note that in any reachable marking, the token on p^* contains the global history of N', while the histories in the tokens of N' are partial suborders of the global history. (N, m_0) and (N', m_0') are bisimilar by construction.

Corollary 16. *Token history nets are Turing complete.*

Proof. By Lemma 15 and Corollary 14.

It is easy to map both a token history net and a global history net onto a colored Petri net with token values being histories. Figure 4 shows a screenshot of CPN Tools simulating the two traffic lights from Example 1 controlled by history. Note that we added place *global* to store the global history.

The remainder of this paper focuses on global history nets.

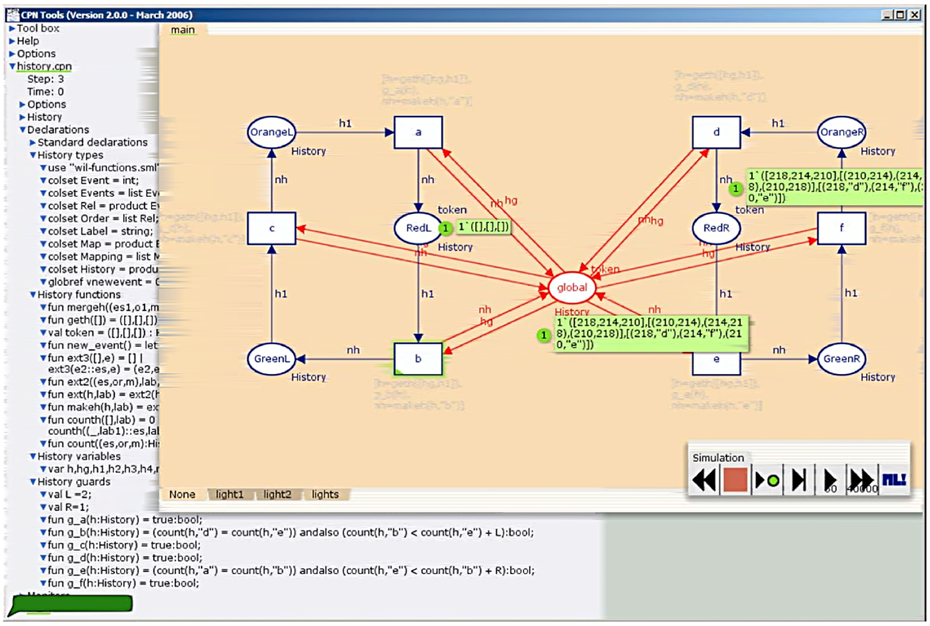
6 Global History Nets with Counting Formulae Guards

In this section we consider global history nets with guards being *counting formulae*, i.e., formulae that do not explore the precedence of events $\prec$. Formally, a *counting formula* φ is defined as

$$\varphi ::= \mathit{false} \mid \varphi \Rightarrow \varphi \mid q < q \mid l == l$$

where q and l are terms and label expressions as in Definition 6.

Note that global history nets with counting formulae guards are Turing complete since they allow zero testing on the marking of a place. To facilitate simulation and validation of these nets, we show that every global history net with counting formulae guards can be transformed into a bisimilar inhibitor net. Furthermore, we identify conditions on the global history net implying that the net can be translated to a bisimilar classical Petri net.



```
CPN Tools (Version 2.0.0 - March 2006)
▸Tool box
▸Help
▸Options
▾history.cpn
    Step: 3
    Time: 0
  ▸Options
  ▸History
  ▾Declarations
    ▸Standard declarations
    ▾History types
      ▾use "wil-functions.sml"
      ▾colset Event = int;
      ▾colset Events = list Ev
      ▾colset Rel = product E
      ▾colset Order = list Rel;
      ▾colset Label = string;
      ▾colset Map = product E
      ▾colset Mapping = list M
      ▾colset History = produ
      ▾globref vnewevent = (
    ▾History functions
      ▾fun mergeh((es1,o1,m
      ▾fun geth([]) = ([],[],[])
      ▾val token = ([],[],[]) : H
      ▾fun new_event() = let
      ▾fun ext3([],e) = [] |
        ext3(e2::es,e) = (e2,e
      ▾fun ext2((es,or,m),lab,
      ▾fun ext(h,lab) = ext2(h
      ▾fun makeh(h,lab) = ex
      ▾fun counth([],lab) = 0
        counth((_,lab1)::es,lal
      ▾fun count((es,or,m):Hi
    ▾History variables
      ▾var h,hg,h1,h2,h3,h4,r
    ▾History guards
      ▾val L =2;
      ▾val R=1;
      ▾fun g_a(h:History) = true:bool;
      ▾fun g_b(h:History) = (count(h,"d") = count(h,"e")) andalso (count(h,"b") < count(h,"e") + L):bool;
      ▾fun g_c(h:History) = true:bool;
      ▾fun g_d(h:History) = true:bool;
      ▾fun g_e(h:History) = (count(h,"a") = count(h,"b")) andalso (count(h,"e") < count(h,"b") + R):bool;
      ▾fun g_f(h:History) = true:bool;
```

main
OrangeL
h1
a
History
nh
c
RedL
token
1`([],[],[])
History
nhhg
global
token
History
1`([218,214,210],[(210,214),(214,218),(210,218)],[(218,"d"),(214,"f"),(210,"e")])
h1
GreenL
nh
b
d
OrangeR
RedR
f
e
GreenR
1`([218,214,210],[(210,214),(214,8),(210,218)],[(218,"d"),(214,"f"),(:0,"e")])
Simulation
None
light1
light2
lights

Fig. 4. The history-dependent Petri net with parameters R en L and using a global place to record history simulated using CPN Tools

6.1 Nets with Counting Formulae as Guards vs. Inhibitor Nets

We start with the simplest form of counting formulae, namely $(\#A)\ \rho\ (\#B + k)$ for some $A, B \subseteq \Sigma$, $\rho \in \{\geq, \leq\}$ and $k \in \mathbb{N}$. For the sake of brevity we call these expressions *basic counting formulae* (over A and B). Note that taking B equal to $\emptyset$ we obtain $(\#A)\ \rho\ k$ (since $\#\emptyset = 0$).

Lemma 17. *Let (N, m_0) be a marked global history net with $N = \langle P, T, F, \Lambda, g\rangle$ such that for any $t \in T$, $g(t)$ is a basic counting formula. There exists a marked inhibitor net (N', m_0') bisimilar to (N, m_0).*

Proof. We apply to the net (N, m_0) an iterative process of guard elimination resulting in (N', m_0'). At every iteration step we will replace one of the transition guards of the current net by *true*, adding some places and transitions to preserve the net behavior. The process terminates when all guards are *true*, i.e. we obtained a regular inhibitor net.

Let t be a transition whose guard we eliminate at the next step and let $g(t)$ be $\#A\ \rho\ \#B + k$ for some $A, B \subseteq \Sigma$, $\rho \in \{\geq, \leq\}$ and $k \in \mathbb{N}$. We can assume that A and B are disjoint, since $(\#A)\ \rho\ (\#B + k)$ if and only if $(\#(A \setminus B))\ \rho\ (\#(B \setminus A) + k)$.

Figure 5 shows the basic idea of the eliminating a transition with guard $g(t)$. Consider, for example the case ρ equals $\leq$. Figure 5(a) sketches the relevant features of the initial net and Figure 5(b) shows the net where guard

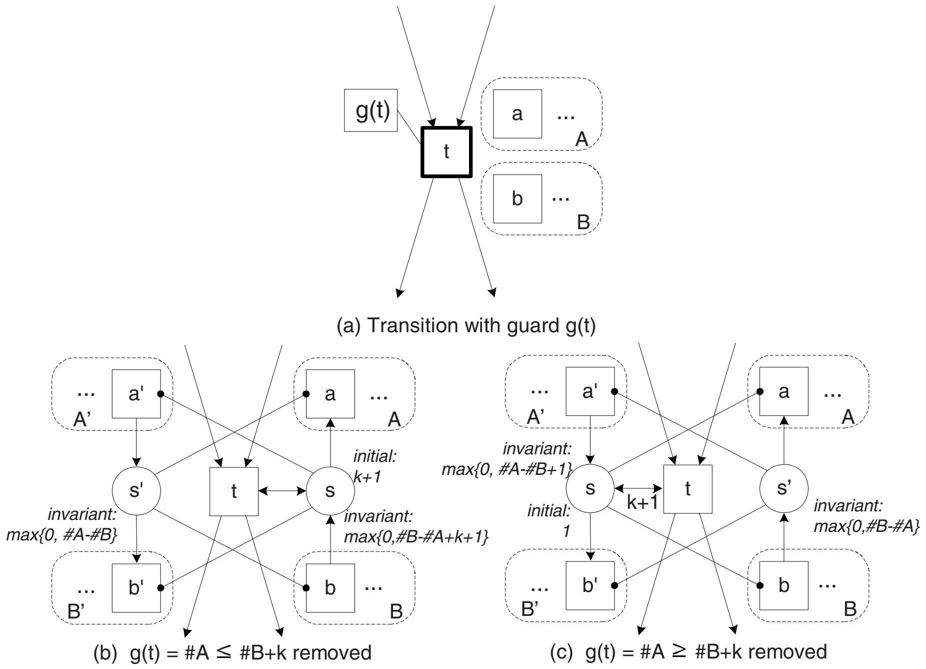


Fig. 5. Replace the guard by places s and s', duplicate transitions, and inhibitor arcs

$g(t) = (\#A \leq \#B + k)$ is eliminated. Note that A and B refer to the sets of transitions having a label from A respectively B. For the purpose of illustration, we show a transition with label $a \in A$ and a transition with label $b \in B$ (note that may not be such transitions).

In order to mimic the guard $g(t)$, we add places s and s', where s will contain $\max\{0, \#B - \#A + k + 1\}$ tokens while s' will contain $\max\{0, \#A - \#B\}$ tokens. Note that $g(t) = (\#A \leq \#B + k)$ evaluates to *true* if and only if there is at least one token in s, therefore we add a bidirectional arc between s and t. In the initial marking $m_0(s) = k + 1$ and $m_0(s') = 0$.

To support the computations on s and s', we need to duplicate all transitions with a label from $A \cup B$, i.e., for every v such that $\Lambda(v) \in A$ or $\Lambda(v) \in B$ we add a transition v' with $^\bullet v' = {}^\bullet v$, $v'^\bullet = v^\bullet$, and $\Lambda(v') = \Lambda(v)$. The resulting sets of transitions are referred to as A' and B' in Figure 5(b). It is essential to note that the transitions are mutually exclusive in terms of enabling and that s and s' are non-blocking, i.e., if $v \in T$ was enabled in the original net, then either v or v' is enabled in the net with inhibitors.

The construction for ρ equal to $\geq$ is similar as shown in Figure 5(c). Note that the initial marking has been updated and that t now tests for the presence of $k + 1$ tokens in s where s always contains $\max\{0, \#A - \#B + 1\}$ tokens.

The transformation is repeatedly applied until no guarded transition is left. The bisimilarity of (N, m_0) and (N', m'_0) can be trivially proven by induction. $\square$

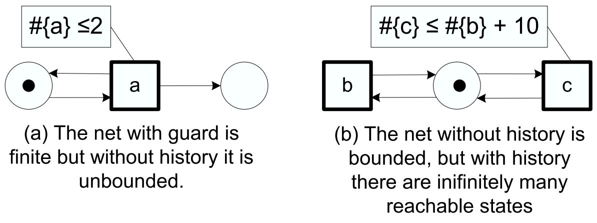

#{a} ≤2
a
(a) The net with guard is finite but without history it is unbounded.
#{c} ≤ #{b} + 10
b
c
(b) The net without history is bounded, but with history there are inifinitely many reachable states

Fig. 6. Bounded and unbounded nets

Our interest in transitions with basic counting formulae as guards is motivated by the fact that any non-trivial counting formula is equivalent to a disjunction of conjunctions of basic counting formulae.

Lemma 18. *Any counting formula φ can be written in disjunctive normal form where the literals are positive basic counting formula (i.e. without negations), so $\varphi \equiv$ true or $\varphi \equiv$ false or $\varphi \equiv \bigvee_i(\bigwedge_j \psi_{i,j})$ and each $\psi_{i,j}$ is a basic counting formula.*

Theorem 19. *Let (N, m) be a marked global history net with $N = \langle P, T, F, \Lambda, g\rangle$ such that for any $t \in T$, $g(t)$ is a counting formula. There exists a marked inhibitor net (N', m') bisimilar to (N, m).*

Proof. (Idea) By Lemma 18 we consider only disjunctions of conjunctions of basic counting formulae. First we transform our net to a net where all guards are conjunctions of basic counting formulae by applying the following construction: Every transition t with a guard $\varphi \vee \psi$ is replaced by transitions t_φ with the guard φ, and t_ψ with the guard ψ, where ${}^\bullet t_\varphi = {}^\bullet t_\psi = {}^\bullet t$, $t_\varphi^\bullet = t_\psi^\bullet = t^\bullet$ and $\Lambda(t_\varphi) = \Lambda(t_\psi) = \Lambda(t)$.

At the next step we eliminate conjuncts from the guards one by one by applying the construction depicted in Figure 5. The only difference is that we apply the construction to a transition t with a guard $(\#A\ \rho\ \#B + k) \wedge \varphi$, and the guard of t in the resulting net is then φ. $\square$

Boundness and analyzability of global history nets. Although the construction referred to in the proof of Theorem 19 is applicable to any global history net with counting formulae as guards, the resulting net contains inhibitor arcs and therefore, cannot be analyzed easily because of Turing completeness. However, it is well-known that inhibitor arcs can be eliminated in *bounded* inhibitor nets. Boundedness of classical or inhibitor Petri nets is in principle finiteness of its state space. Hence it is interesting to explore "finiteness notions" for global history nets.

Finiteness of $\mathcal{R}_N((m_0, \epsilon))$ for a global history net $N = (\langle P, T, F, \Lambda, g\rangle)$ does not imply boundedness of the underlying Petri net $(\langle P, T, F, \Lambda\rangle, m_0)$ and vice versa. In Figure 6 we see two global history nets. The underlying Petri net shown in Figure 6(a) is unbounded, while the global history net has a finite state space due to the transition guard. The underlying Petri net shown in Figure 6(b) is bounded, while the global history net has an infinite state space just because it has an unbounded history. Still, the behavior of this net is clearly analyzable, since it is possible to construct a classical Petri net bisimilar to it. The latter observation motivates our interest in the existence of a classical Petri net bisimilar to a global history net.

In the two following subsections we discuss sufficient conditions for the existence of a bisimilar classical Petri net.

6.2 Guards Depending on the Marking Only

In this subsection we give conditions on the guards that allow a transformation into an equivalent bounded Petri net. So global history nets satisfying these conditions will accept regular languages. We consider here guards that depend only on the marking. As stated by Proposition 13 if transitions have unique labels, then a marking is uniquely determined by the history.

Definition 20. *Given a global history net $N = \langle P, T, F, \Lambda, g\rangle$ with Λ being injective, we say that a formula φ is a* marking formula *if there exists a formula ψ, $\varphi \equiv \psi$, such that ψ is a counting formulae and every basic counting formula in ψ is of the form $(\#^\bullet p)\ \rho\ (\#p^\bullet + k)$ for $p \in P$ or $(\#^\bullet p + k)\ \rho\ (\#p^\bullet)$, $k \in \mathbb{N}$ and $\rho \in \{\le, \ge\}$.*

Theorem 21. *Let $N = \langle P, T, F, \Lambda, g\rangle$ be a global history net with injective Λ such that for any $t \in T$, $g(t)$ is a marking formula. If the underlying Petri net $(\langle P, T, F, \Lambda\rangle, m_0)$ is bounded, then there exists a bounded marked Petri net bisimilar to $(N, (m_0, \epsilon))$.*

Proof (Idea). We construct a net $N'' = \langle P', T', F'', \Lambda\rangle$ and a marking m_0'' such that (N'', m_0'') bisimilar to (N, m_0). We start by adding a duplicate place p' for every place $p \in P$ such that ${}^\bullet p' = p^\bullet$ and $p'^\bullet = {}^\bullet p$. Since the underlying Petri net is bounded, there exists $b \in \mathbb{N}$ such that for any reachable marking m and any place p, $m(p) \le b$. We take n greater than the sum of b and the maximum of all constants in the guards. We define m_0' for N' as follows: $\forall p \in P : m_0'(p) = m_0(p) \wedge m_0'(p') = n - m_0(p)$. Observe that $m(p) + m(p') = n$ for any reachable marking m. Moreover, by construction, $\#^\bullet p = \#p'^\bullet$ and $\#p^\bullet = \#^\bullet p'$ for any place p.

Without loss of generality we assume that transition guards are conjunctions of the form $(\#^\bullet p)\ \rho\ (\#p^\bullet + k)$ with $k \ge 0$ and $\rho \in \{\le, \ge\}$. Indeed, first, the proof of Theorem 19 shows how general counting formulae can be reduced to basic counting formula. Second, if the guard is of the form $(\#^\bullet p + k)\ \rho\ \#p^\bullet$, by the previous observation, we obtain $(\#p'^\bullet + k)\ \rho\ \#^\bullet p'$, i.e., $(\#^\bullet p')\ \rho'\ (\#p'^\bullet + k)$ with ρ' being the comparison dual to ρ, i.e. $\rho' \in \{\le, \ge\} \setminus \{\rho\}$. We denote the resulting net $N' = \langle P', T', F', \Lambda\rangle$. Next we are going to add arcs depending on the guards of N.

We distinguish between two cases. Let $g(t)$ be $(\#^\bullet p) \leq (\#p^\bullet + k)$. Then t may fire only if the number of tokens consumed from p does not exceed the number of tokens produced to p by more than k, i.e., the number of tokens produced to p' does not exceed the number of tokens consumed from p' by more than k. In other words, $m_0'(p')$ has at least k tokens. Moreover, if $t \in {}^\bullet p'$ then t may fire only if p' contains at least $F'(p,t)$ tokens. Therefore, we add an arc between p' and t: $F''(p',t) = \max\{F'(p',t), m_0'(p') - k\}$, i.e., $\max\{F(t,p), n - k - m_0(p)\}$. To complete the transformation, observe that we are not allowed to change the behavior of the original net. Thus, we need to *return* tokens to p'. To this end we add an arc between t and p': $F''(t,p') = F'(t,p') + max\{0, m_0'(p) - k - F'(p',t)\}$, i.e., $F(p,t) + max\{0, n - k - m_0(p) - F(t,p)\}$.

Observe that this case also covers the situation when $g(t)$ is $(\#^\bullet p) \geq (\#p^\bullet + k)$ and $k = 0$. Therefore, we assume in the second case $((\#^\bullet p) \geq (\#p^\bullet + k))$ that $k > 0$. Similarly to the previous case, we add two arcs between p and t: $F''(p,t) = \max\{F'(p,t), k + m_0'(p)\}$, i.e., $\max\{F(p,t), k + m_0(p)\}$, and $F''(t,p) = F'(t,p) + \max\{0, k + m_0'(p) - F'(p,t)\}$, i.e., $F(t,p) + \max\{0, k + m_0(p) - F(p,t)\}$.

In both cases t can fire if and only if the guard holds and the firing does not change the behavior of the original net. □

6.3 Counting Formulae with Bounded Synchronization Distance

In this subsection we consider a condition on guards that allows to transform a global history net to a bisimilar Petri net, which is not necessarily bounded. We use here an important concept in Petri nets introduced by Carl Adam Petri: *synchronization distance* [4,7,13]. We use a generalization of this notion, the so-called *y-distance* [16].

Definition 22. *Let (N, m_0) be a Petri net and n be the number of transitions in N. For a* weight vector $y \in \mathbb{Z}^n$ *the y*-distance *of (N, m_0) is defined by*

$$D((N, m_0), y) = \sup_{\sigma \in \Delta} y^T \cdot \overline{\sigma},$$

where y^T is the transpose of y, $\overline{\sigma}$ is the Parikh vector of σ and Δ the set of all executable finite firing sequences. The synchronization set *is*

$$Sync((N, m_0)) = \{y \in \mathbb{Z}^n \mid D((N, m_0), y) < \infty\}.$$

In the net on the right-hand of Figure 6 transitions b and c can fire infinitely often. If we take the underlying classical Petri net, the transitions are completely independent of each other and the y-distance is ∞ for any weight vector with at least one positive component. If we consider the global history net instead, the number of the firings of c never exceeds the number of the firings of b by more then 11. Hence the y-distance with $y = \langle -1, 1 \rangle$ is 11. On the other hand, the number of firings of b is not restricted by the number of the firings of c, and the y-distance for $y = \langle 1, -1 \rangle$ is ∞.

For two label sets A and B, the *characteristic weight vector for (A,B)*, denoted $y_{(A,B)}$, is the weight vector with components equal to 1 for transitions with labels

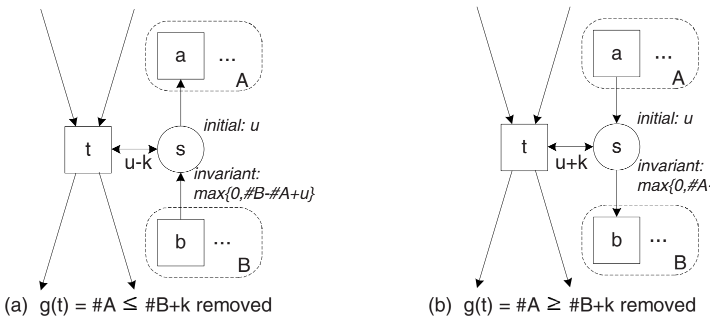

a
...
A
t
u-k
s
initial: u
invariant: max{0,#B-#A+u}
b
...
B
(a) g(t) = #A ≤ #B+k removed
a
...
A
t
u+k
s
initial: u
invariant: max{0,#A-#B+u}
b
...
B
(b) g(t) = #A ≥ #B+k removed

Fig. 7. Transforming nets with synchronization distance restrictions

in A, -1 for transitions with labels in B and 0 for all other vector components (recall that we may safely assume that A and B are disjoint). We denote the $y_{(A,B)}$-distance by $d(A,B)$ and we call it the *characteristic distance (A,B)*. In [16], an algorithm is given to decide whether $y \in Sync((n, m_0))$ and to determine the y-distance by examining a finite set of vectors.

Theorem 23. *Let $N = \langle P, T, F, \Lambda, g\rangle$ be a global history net with initial marking m_0 such that for any $t \in T$, $g(t)$ is a disjunction of conjunctions of counting formulae of the form $\#A \;\rho\; \#B + k$ with $\rho \in \{<, \leq, >, \geq\}$, for each of which the following property holds: if ρ is $\leq$ then $d(A,B) < \infty$ and if ρ is $\geq$ then $d(B,A) < \infty$ in the underlying Petri net $(\langle P, T, F, \Lambda\rangle, m_0)$. Then there exists a marked Petri net (N', m'_0) bisimilar to (N, m_0).*

Proof. (Idea) The proof is done by construction. Disjunctions and conjunctions are taken care of as in Theorem 19. Therefore, we restrict our attention to the following special case: the guard of transition t is a basic counting formula of the form $\#A \;\rho\; \#B + k$.

For the first case, where ρ is $\leq$, we set $u = \max\{k, d(A,B)\} + 1$. Note that $u \leq k$ implies that the guard of t will always be evaluated to *true*, and thus may be trivially removed. So we assume that $u > k$. We apply the construction shown at the left-hand side of Figure 7. A new place s is added with $F(b,s) = 1$ for all b such that $\Lambda(b) \in B$, $F(s,a) = 1$ for all a such that $\Lambda(a) \in A$, and $F(s,t) = F(t,s) = u - k$. Furthermore, the initial marking is $m'_0(s) = u$. Transition t can fire if and only if s contains at least $u - k$ tokens. Note that $u - k > 0$ and that for any reachable state (m, H) we have $m'(s) = u + \#B - \#A \geq u - d(A,B) > 0$. Therefore t can fire only if $\#B - \#A \geq -k$ and the transitions with labels in A or B are thus not restricted in their firings.

The second case, displayed in the right-hand net of Figure 7, is similar: $u = \max\{k, d(B,A)\} + 1$, the arcs are reversed, $F(s,t) = F(t,s) = u + k$ and $m'_0(s) = u$. $\square$

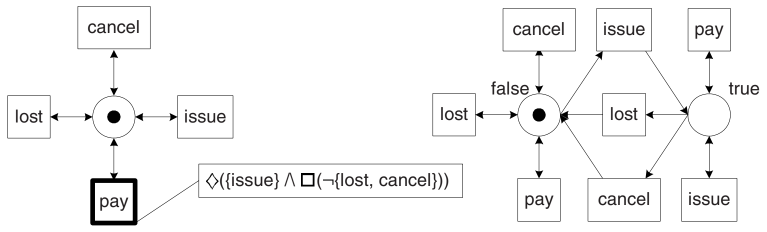


Fig. 8. A net with an LTL-guard

7 Global History Nets with LTL Guards

Now we consider the class of global history nets with LTL guards. We consider the next-free variant of LTL, since the next time operator (X) should better be avoided in a distributed context and it is not robust w.r.t. refinements. LTL-formulae are defined by $\phi ::= \mathit{false} \mid \phi \Rightarrow \phi \mid \phi\, U \phi \mid A$, where $A \subseteq \Sigma$ and U is the temporal operator *until*.

Standard LTL works on infinite traces, while our history is always finite. Therefore, we interpret formulae on a trace we observed so far. Let $H = \langle e_1 \ldots e_n \rangle$ be a global history. We define $A(e_i)$ as $(\lambda(e_i) \in A)$, and $(\phi\, U \xi)(e_i)$ as $\exists e_k : ((e_i \prec e_k) \wedge \xi(e_k) \wedge \forall e_m : ((e_i \preceq e_m) \wedge (e_m \prec e_k)) \Rightarrow \phi(e_m))$. We say that $H \models \phi$ iff $H \models \phi(e)$, i.e., $\forall e : ((\forall e_i : e \prec e_i) \Rightarrow \phi(e))$ is evaluated to true. Due to the finiteness of the formula, every LTL formula can be rewritten to a finite formula in our logic. Note that our interpretation of U coincides with the standard one.

Based on the temporal operator U we introduce additional temporal operators $\Diamond$ ("eventually") and $\Box$ ("always") in the standard way: $\Diamond\phi := \mathit{true}\, U \phi$, $\Box\phi := \neg(\Diamond\neg\phi)$.

We will show now how to translate a global history net with LTL guards to a (classical) Petri net.

While LTL formulae over infinite traces can be translated to Büchi automata, LTL formulae over finite traces can be translated to finite automata. [5] presents a translation algorithm that modifies standard LTL to Büchi automata conversion techniques to generate finite automata that accept finite traces satisfying LTL formulae. The main aspect of modification there is the selection of accepting conditions. The automata generated are finite automata on finite words. Therefore, they can be made deterministic and minimized with standard algorithms [10].

Let $N = \langle P, T, F, \Lambda, g \rangle$ be a given global history net. At the first step of our transformation we build a finite deterministic automaton whose edges are labeled by action names from $\Lambda(T)$ for every non-trivial (not *true* or *false*) transition guard. Then we transform this automaton into a marked Petri net (which is a state machine) where a token is placed on the place corresponding to the initial state of the automaton, and final places obtain auxiliary labels *true* and non-final places are labeled by *false*.

Fig. 8 shows a simplistic example for a credit card company, where a credit card can be issued, reported lost, used for a payment or cancelled. The payment transition *pay* has a guard requiring that the payment is possible only if the card has not being lost or cancelled after its last issue ($\Diamond(\{issue\} \wedge \Box(\neg\{lost, cancel\}))$. The net corresponding to the guard is shown on the right hand side of the figure. Note that this net can perform an arbitrary sequence of steps, and the place "*true*" has a token when the guard on the history should be evaluated to *true* and "*false*" when the guard should be evaluated to *false*.

At the next step we build the net N_S which is a synchronous product of the Petri net $N_P = \langle P, T, F, \Lambda \rangle$ with the guard nets N_i, by synchronizing on transition labels. Namely, the set of places P_S of the synchronous product is the union $P \cup (\cup_i P_i)$ of places of N and the places of the guard nets; every combination of transitions $t, t_1, \ldots, t_n$, where $t \in T, t_i \in T_i$ and $\Lambda(t) = \Lambda_i(t_i)$ for all i, is represented in T_S by a transition t' with ${}^\bullet t' = {}^\bullet t + \sum_i {}^\bullet t_i,\ t'^\bullet = t^\bullet + \sum_i t_i^\bullet$ and $\Lambda(t') = \Lambda(t)$.

The guard nets can perform any arbitrary sequence from Σ^* and N_S has thus the same behavior as N_P. Now we restrict the behavior of N_S by taking the evaluations of guards into account. To achieve it, we add biflow arcs between every transition $t \in N_S$ and every *true*-place corresponding to the guard net of this transition. The obtained net is bisimilar to the original global history net by construction.

8 Related Work

Histories and related notions such as event systems [19] and pomsets [8,3] have been used in the past to provide causality-preserving semantics for Petri nets. Unlike our approach, these works did not aim at restricting the firings by means of history-dependent guards. Baldan *et al.* [2] use two different notions of history. First of all, they consider *semi-weighted nets*, i.e., nets where every token can be uniquely identified by means of tokens used to produce it, transition that produces it and the name of the place where it resides. This idea is similar in spirit to our token history. However, the authors do not make this notion of history explicit nor do they discuss additional operations that can be performed on histories. Neither this notion, nor history as configuration used by the authors in study of causality, can be used to restrict firings of transitions by means of guards as suggested in our approach.

History-dependent automata [12] extend states and transitions of an automaton with sets of local names: each transition can refer to the names associated to its source state but can also generate new names which can then appear in the destination state. This notion of history implies that one cannot refer to firings of other transitions but by means of shared names. We believe that the ability to express dependencies on previous firings explicitly is the principal advantage of our approach.

Operations on pomsets similar to our union and intersection appeared under different names in [6,14,18]. The major distinction is due to unimportance of the events' identities in these approaches. Therefore, these operations make use of disjoint sum to define a union and bijectively rename the events to define an intersection. Therefore, these operations are defined for any pomsets. Unlike the existing approaches, we take the identities of the events into account. This guarantees that common parts of histories appear only once in their union, and only truly common events appear in the intersection.

y-distance and related notions were studied starting from [4,7,13,16]. Silva and Murata [15] introduced group-B-fairness, where they extend the synchronization distance notion from single transitions to the groups of transitions, like we do in Subsection 6.3. The focus of Silva and Murata's paper is however on *group-B-fair nets*, i.e., nets such that any pair of transition sets from a given transition covering is in a group-B-fair relation. Unlike their work, Theorem 23 demands being in a group-B-fair relation only for sets of transitions corresponding to sets of labels used in the guards.

9 Conclusion

In this paper we emphasize the importance of taking history into account while modelling processes. Historical information is present in most state-of-the-art enterprise information systems. Moreover, it allows to separate process information from safety constraints, improving the readability and maintainability of models.

We have provided means to model history-dependent processes by extending the classical Petri nets model and considered two ways of incorporating history: token history nets and global history nets. To provide analysis, simulation and validation facilities, we have put a link from global history nets to classical and inhibitor Petri nets. Namely, we have identified several subclasses of global history nets that can be automatically transformed to classical Petri nets. For the class of global history nets with counting formulae as guards we have defined a transformation to inhibitor nets. Finally, observe that global history nets can be easily implemented in CPN Tools [1].

Future work. For the future work we plan to adapt our token net framework for modelling component-based systems. We intend to extend the language of operations on histories by adding projection in order to allow information hiding and intersection to check disjointness/presence of common parts in token histories. The guard language will allow to evaluate conditions both on separate tokens and on their combinations.

We are going to develop a method for transforming broader subclasses of global history nets to classical and inhibitor Petri nets. For instance, our transformation of global history nets with LTL guards can be easily extended for LTL with Past. We also consider developing a transformation for global history nets with LogLogics [9] guards, a three-valued variant of LTL+Past on finite traces.

Acknowledgement. We are grateful to Jan Hidders and Jan Paredaens for a number of fruitful discussions at the early stages of this research.

References

1. CBN Tools http://wiki.daimi.au.dk/cpntools/cpntools.wiki
2. Baldan, P., Busi, N., Corradini, A., Pinna, G.M.: Domain and event structure semantics for Petri nets with read and inhibitor arcs. Theoretical Computer Science 323(1-3), 129–189 (2004)
3. Best, E., Devillers, R.R.: Sequential and concurrent behaviour in Petri net theory. Theoretical Computer Science 55(1), 87–136 (1987)
4. Genrich, H.J., Lautenbach, K., Thiagarajan, P.S.: Elements of general net theory. In: Proceedings of the Advanced Course on General Net Theory of Processes and Systems, London, UK, pp. 21–163. Springer, Heidelberg (1980)
5. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: ASE, IEEE Computer Society (Full version available as a technical report) pp. 412–416 (2001)
6. Gischer, J.L.: The equational theory of pomsets. Theoretical Computer Science 61, 199–224 (1988)
7. Goltz, U., Reisig, W.: Weighted Synchronic Distances. In: Girault, C., Reisig, W. (eds.) Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets. Informatik-Fachberichte, vol. 52, pp. 289–300. Springer, Heidelberg (1981)
8. Goltz, U., Reisig, W.: The non-sequential behavior of Petri nets. Information and Control 57(2/3), 125–147 (1983)
9. van Hee, K., Oanea, O., Serebrenik, A., Sidorova, N., Voorhoeve, M.: LogLogics: A logic for history-dependent business processes, vol. 65(1) (2007)
10. Hopcroft, J., Ullman, J.: Introduction to Automata, Theory, Languages, and Computation. Addison-Wesley, London (1979)
11. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. In: Monographs in Theoretical Computer Science, Springer, Heidelberg (1997)
12. Montanari, U., Pistore, M.: History-dependent automata: An introduction. In: Bernardo, M., Bogliolo, A. (eds.) SFM-Moby 2005. LNCS, vol. 3465, pp. 1–28. Springer, Heidelberg (2005)
13. Petri, C.A.: Interpretations of net theory. Technical Report ISF-Report 75.07 (1975)
14. Pratt, V.R.: Some constructions for order-theoretic models of concurrency. In: Parikh, R. (ed.): Logics of Programs. LNCS, vol. 193, pp. 269–283. Springer, Heidelberg (1985)
15. Silva, M., Murata, T.: B-fairness and structural b-fairness in Petri net models of concurrent systems. J. Comput. Syst. Sci. 44(3), 447–477 (1992)
16. Suzuki, I., Kasami, T.: Three measures for synchronic dependence in Petri nets. Acta Inf. 19, 325–338 (1983)
17. Valk, R.: On the computational power of extended Petri nets. In: Winkowski, J. (ed.): Mathematical Foundations of Computer Science 1978. LNCS, vol. 64, pp. 526–535. Springer, Heidelberg (1978)
18. Wimmel, H., Priese, L.: Algebraic characterization of Petri net pomset semantics. In: Mazurkiewicz, A.W, Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 406–420. Springer, Heidelberg (1997)
19. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1986)

Complete Process Semantics for Inhibitor Nets

Gabriel Juhás[2], Robert Lorenz[1], and Sebastian Mauser[1]

[1] Department of Applied Computer Science,
Catholic University of Eichstätt-Ingolstadt
{robert.lorenz,sebastian.mauser}@ku-eichstaett.de
[2] Faculty of Electrical Engineering and Information Technology
Slovak University of Technology, Bratislava, Slovakia
gabriel.juhas@stuba.sk

Abstract. In this paper we complete the semantical framework proposed in [13] for process and causality semantics of Petri nets by an additional aim and develop process and causality semantics of place/transition Petri nets with weighted inhibitor arcs (pti-nets) satisfying the semantical framework including this aim. The aim was firstly mentioned in [8] and states that causality semantics deduced from process nets should be *complete* w.r.t. step semantics in the sense that *each* causality structure which is consistent with the step semantics corresponds to some process net. We formulate this aim in terms of *enabled* causality structures.

While it is well known that process semantics of place/transition Petri nets (p/t-nets) satisfy the additional aim, we show that the most general process semantics of pti-nets proposed so far [13] does not and develop our process semantics as an appropriate generalization.

1 Introduction

The study of concurrency as a phenomenon of system behavior attracted much attention in recent years. There is an increasing number of distributed systems, multiprocessor systems and communication networks, which are concurrent in their nature. An important research field is the definition of non-sequential semantics of concurrent system models to describe concurrency among events in system executions, where events are considered concurrent if they can occur at the same time and in arbitrary order. Such non-sequential semantics is usually deduced from the so called step semantics of a concurrent system model.

For the definition of step semantics it is generally stated which events can occur in a certain state of the system *at the same time* (synchronously) and how the system state is changed by their occurrence. Such events form a *step (of events)*. Given an initial state, from this information all sequences of steps which can occur from the initial marking can easily be computed. The set of all possible such *step sequences* defines the step semantics of a concurrent system model. A step sequence can be interpreted as a possible *observation* of the systems behavior, where the event occurrences in one step are observed at the same time and the event occurrences in different steps are observed in the order given by the step sequence.

Non-sequential semantics are based on causal structures – we will also call them scenarios in the following – which allow to specify arbitrary concurrency relations among events. Non-sequential semantics for this paper is a set of scenarios. A scenario allows (generates) several different observations, since the occurrence of events which are concurrent in the scenario can be observed synchronously or also in arbitrary order. Therefore, a given scenario only represents behavior of the system if it is consistent with the step semantics in the sense that all of its generated observations belong to the step semantics of the system. Non-sequential semantics which consists only of scenarios satisfying this property we call *sound w.r.t. step semantics*. On the other hand, *all* scenarios which are consistent with the step semantics represent behavior of the system. Non-sequential semantics which contains *all* such scenarios we call *complete w.r.t. the step semantics*. In other words, a complete non-sequential semantics includes each causal structure satisfying that all observations generated by the causal structure are possible observations of the system. Note here that if we add causality to a causal structure which is consistent with the step semantics the resulting causal structure is again consistent with the step semantics (since it generates less observations). Thus, a complete non-sequential semantics can be given by such causal structures consistent with the step semantics satisfying that removing causality from the causal structure results in a causal structure *not* consistent with the step semantics. Such causal structures express *minimal* causal dependencies among events. Altogether, complete non-sequential semantics represent minimal causalities.

Therefore, an important aim of each semantical framework for the definition of a non-sequential semantics of particular formalisms for concurrent systems is that a non-sequential semantics is defined *sound and complete w.r.t. the step semantics* of the formalism. In this paper we consider this aim for Petri nets. These are one of the most prominent formalisms for understanding the concurrency phenomenon on the theoretical as well as the conceptual level and for modeling of real concurrent systems in many application areas [7]. The most important and well-known concept of non-sequential semantics of Petri nets are process semantics based on occurrence nets [4,5]. From the very beginning of Petri net theory processes were based on partial orders relating events labeled by transitions (an event represents the occurrence of a transition): Any process directly defines a respective partial order among events, called the associated *run*, in which unordered events are considered to be concurrent. Since adding causality to a run still leads to possible system behavior, a non-sequential semantics of a Petri net can also be given as the set of sequentializations of runs (a sequentialization adds causality) of the net. This set is also called causal semantics of the net, since it describes its causal behavior. Note that in most cases partial orders are suitable to describe such behavior but sometimes generalizations of partial orders are needed as appropriate causal structures. In the case of inhibitor nets under the so-called a-priori semantics [6], so called stratified order structures (so-structures) represent the causal semantics.

Since the basic developments of Petri nets, more and more different *Petri net classes* for various applications have been proposed. It turned out to be not easy to define process semantics and related causality semantics in the form of runs for such net classes. Therefore, in [13] (in the context of defining respective semantics for inhibitor nets) a semantical framework aiming at a systematic presentation of process and causality semantics of different Petri net models was developed (see Figure 3 in Section 3): Any process semantics should fulfill the reasonable aims stated by the framework.

These aims are reduced to several properties that have to be checked in a particular practical setting. The most important of these aims is the soundness of process semantics and causality semantics w.r.t. step semantics as described above. For Petri nets, soundness means that each observation generated by a process or a run is a possible step occurrence sequence of the Petri net. But this general framework – as well as many other particular process definitions for special Petri net classes – does not regard the described aim of completeness. In the Petri net context, process and causality semantics are complete w.r.t. step semantics if each causality structure consistent with the step semantics adds causality to or is equal to some run of the Petri net. Instead another aim of the framework from [13] requires a kind of weak completeness, saying that each step occurrence sequence should be generated by some process.

For place/transition nets (p/t-nets) a labeled partial order (LPO) which is consistent with the step semantics is called *enabled* [17,18,8]. It was shown in [11] that an LPO is enabled if and only if it is a sequentialization of a run corresponding to a process (see also [17,18,8]). Thus, process and causality semantics of p/t-nets are sound and complete w.r.t. step semantics. In particular, from the completeness we deduce that enabled LPOs with minimal causal dependencies between events (thus maximal concurrency) – so called *minimal enabled LPOs* – are generated by processes.[1] This is an essential property of p/t-net processes and justifies their success as non-sequential semantics describing system behavior.

Therefore, the aim of completeness should also hold for process semantics of other Petri net classes. To this end, we included it in the semantical framework of [13]. We will discuss the aim of completeness for process definitions of inhibitor nets. As stated in [15], ”Petri nets with inhibitor arcs are intuitively the most direct approach to increasing the modeling power of Petri nets”. Moreover inhibitor nets have been found appropriate in various application areas [1,3]. Accordingly, for these net classes various authors proposed process definitions regarding different interpretations of the occurrence rule of inhibitor nets. In this paper we will focus on the most general class of pti-nets and its process definition from [13].[2] We show that the general a-priori process definition of [13] does not fulfill the aim of completeness and propose appropriate changes of the process semantics. Thus we develop an alternative process definition which fulfills the complete semantical framework of Figure 3 including the aim of completeness.

As mentioned in the context of the a-priori semantics, LPOs are not expressive enough to describe the causal behavior of a pti-net. Instead, so-structures are used on the causal level. Thus the aim of completeness can be formulated for this net class in the following way: For any enabled so-structure there is a process with associated run in the form of an so-structure such that the enabled so-structure sequentializes the run. As in the case of LPOs, an so-structure is enabled if it is consistent with the step semantics of pti-nets in the above described sense.

The paper is structured as follows: First the basic notions of pti-nets, processes of pti-nets, so-structures (see [13]) and enabled so-structures are introduced (section 2). Then in section 3 the semantical framework of [13] will be discussed in the context of

[1] In case of p/t-nets and their processes (runs), not each enabled LPO is a run and there are also non-minimal runs, but each minimal enabled LPO is a minimal run.

[2] We will briefly consider alternative process definitions for inhibitor nets in the conclusion.

introducing a new requirement – the aim of completeness. Subsequently in the main part of the paper (section 4) we will show why the a-priori process semantics for pti-nets in [13] does not fulfill the aim of completeness. Based on these considerations we propose an alternative process semantics implementing the complete semantical framework including the aim of completeness.

2 Preliminaries

In this section we recall the basic definitions of *so-structures*, *pti-nets (equipped with the a-priori semantics)* and *process nets of pti-nets*, and finally define *enabled so-structures*.

Given a set X we will denote the set of all subsets of X by 2^X and the set of all multi-sets over X by $\mathbb{N}^X$. A set can always be viewed as a multi-set m with $m \leq 1$ and correspondingly a multi-set $m \leq 1$ can always be viewed as a set. We further denote the identity relation over X by id_X, the reflexive, transitive closure of a binary relation R over X by R^*, the transitive closure of R by R^+ and the composition of two binary relations R, R' over X by $R \circ R'$.

Inhibitor nets are an extension of classical Petri nets enhanced with inhibitor arcs. In their simplest version inhibitor arcs test whether a place is empty in the current marking (zero-testing) as an enabling condition for transitions. In the most general version of pti-nets, inhibitor arcs test if a place contains *at most* a certain number of tokens given by weights of the inhibitor arcs (instead of zero-testing). In pictures inhibitor arcs are depicted by arcs with circles as arrowheads. Figure 1 shows a pti-net, where the transitions t and v test a place to be empty and transition w tests a place to hold at most one token. As explained in [6,12,13], "earlier than" causality expressed by LPOs is not enough to describe causal semantics of pti-nets w.r.t. the a-priori semantics. In Figure 1 this phenomenon is depicted: In the a-priori semantics the testing for absence of tokens (through inhibitor arcs) precedes the execution of a transition. Thus t cannot occur later than u, because after the occurrence of u the place connected with t by an inhibitor arc (with weight 0 representing zero-testing) is marked. Consequently the occurrence of t is prohibited by this inhibitor arc. Therefore t and u cannot occur concurrently or sequentially in order $u \to t$. But they still can occur synchronously or sequentially in order $t \to u$, because of the occurrence rule "testing before execution" (details on the occurrence rule can be found later on in this section). This is exactly the behavior described by "t not later than u". After firing t and u we reach the marking in which every non-bottom and non-top place of the net NI contains one token. With the same arguments as above the transitions v and w can occur in this marking synchronously but not sequentially in any order. The relationship between v and w can consequently be expressed by a symmetric "not later than" relation between the respective events - none may occur later than the other. The described causal behavior of NI is illustrated through the run $\kappa(\text{AON})$ on the right side of Figure 1. The solid arcs represent a (common) "earlier than" relation. Those events can only occur in the expressed order but not synchronously or inversely. Dashed arcs depict the "not later than" relation explained above. Partial orders can only model the "earlier than" relation, but it is not possible to describe relationships as in the example between t and u as well as between v and w, where synchronous occurrence is possible but concurrency is not existent.

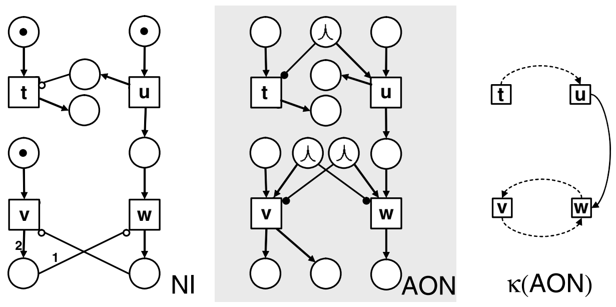


Fig. 1. A pti-net NI (inhibitor arcs have circles as arrowheads), an a-process AON of NI and the associated run $\kappa(\text{AON})$

Altogether there exist net classes including inhibitor nets where synchronous and concurrent behavior has to be distinguished.[3] In [6] causal semantics based on so-structures (like the run $\kappa(\text{AON})$) consisting of a combination of an "earlier than" and a "not later than" relation between events were proposed to cover such cases.

Before giving the definition of *stratified order structures* (*so-structures*), we recall the notion of a *directed graph*. This is a pair $(V, \rightarrow)$, where V is a finite *set of nodes* and $\rightarrow \subseteq V \times V$ is a binary relation over V called the *set of arcs*. Given a binary relation $\rightarrow$, we write $a \rightarrow b$ to denote $(a, b) \in \rightarrow$. Two nodes $a, b \in V$ are called *independent* w.r.t. $\rightarrow$ if $a \not\rightarrow b$ and $b \not\rightarrow a$. We denote the set of all pairs of nodes independent w.r.t. $\rightarrow$ by $\text{co}_{\rightarrow} \subseteq V \times V$. A *(strict) partial order* is a directed graph $\text{po} = (V, <)$, where $<$ is an irreflexive and transitive binary relation on V. If $\text{co}_{<} = id_V$ then $(V, <)$ is called *total*. Given two partial orders $\text{po}_1 = (V, <_1)$ and $\text{po}_2 = (V, <_2)$, we say that po_2 is a *sequentialization* (or *extension*) of po_1 if $<_1 \subseteq <_2$.

So-structures are, loosely speaking, combinations of two binary relations on a set of events where one is a partial order representing an "earlier than" relation and the other represents a "not later than" relation. Thus, so-structures describe finer causalities than partial orders. Formally, so-structures are relational structures satisfying certain properties. A *relational structure* (*rel-structure*) is a triple $\mathcal{S} = (V, \prec, \sqsubset)$, where V is a set (of *events*), and $\prec \subseteq V \times V$ and $\sqsubset \subseteq V \times V$ are binary relations on V. A rel-structure $\mathcal{S}' = (V, \prec', \sqsubset')$ is said to be an *extension* (or sequentialization) of another rel-structure $\mathcal{S} = (V, \prec, \sqsubset)$, written $\mathcal{S} \subseteq \mathcal{S}'$, if $\prec \subseteq \prec'$ and $\sqsubset \subseteq \sqsubset'$.

Definition 1 (Stratified order structure). *A rel-structure* $\mathcal{S} = (V, \prec, \sqsubset)$ *is called* stratified order structure *(*so-structure*) if the following conditions are satisfied for all* $u, v, w \in V$*:*

$$(C1)\, u \not\sqsubset u. \qquad (C3)\, u \sqsubset v \sqsubset w \wedge u \neq w \Longrightarrow u \sqsubset w.$$
$$(C2)\, u \prec v \Longrightarrow u \sqsubset v. \qquad (C4)\, u \sqsubset v \prec w \vee u \prec v \sqsubset w \Longrightarrow u \prec w.$$

In figures, $\prec$ is graphically expressed by solid arcs and $\sqsubset$ by dashed arcs. According to (C2) a dashed arc is omitted if there is already a solid arc. Moreover, we omit arcs which

[3] Further examples of such net classes are briefly mentioned in the conclusion.

can be deduced by (C3) and (C4). It is shown in [6] that $(V, \prec)$ is a partial order (thus a partial order can always be interpreted as an so-structure with $\sqsubset\ = \ \prec$). Therefore, so-structures are a generalization of partial orders. They turned out to be adequate to model the causal relations between events of complex systems regarding sequential, concurrent and synchronous behavior. In this context $\prec$ represents the ordinary "earlier than" relation (as in partial order based systems) while $\sqsubset$ models a "not later than" relation (recall the example of Figure 1).

Similar to the notion of the transitive closure of a binary relation the *$\diamondsuit$-closure* $\mathcal{S}^{\diamondsuit}$ of a rel-structure $\mathcal{S} = (V, \prec, \sqsubset)$ is defined by $\mathcal{S}^{\diamondsuit} = (V, \prec_{\mathcal{S}^{\diamond}}, \sqsubset_{\mathcal{S}^{\diamond}}) = (V, (\prec \cup \sqsubset)^* \circ \prec \circ (\prec \cup \sqsubset)^*, (\prec \cup \sqsubset)^* \setminus id_V)$. A rel-structure $\mathcal{S}$ is called *$\diamondsuit$-acyclic* if $\prec_{\mathcal{S}^{\diamond}}$ is irreflexive. The *$\diamondsuit$-closure* $\mathcal{S}^{\diamondsuit}$ of a rel-structure $\mathcal{S}$ is an so-structure if and only if $\mathcal{S}$ is $\diamondsuit$-acyclic (for this and further results on the $\diamondsuit$-closure see [6]).

For our purposes we will only consider *labeled so-structures* (*LSOs*). Nodes of an LSO represent transition occurrences of a Petri net (constituted by node labels as in Figure 1). Formally LSOs are so-structures $\mathcal{S} = (V, \prec, \sqsubset)$ together with a *set of labels* T and a *labeling function* $l : V \to T$. A labeling function l is lifted to a subset Y of V in the following way: $l(Y)$ is the multi-set over T given by $l(Y)(t) = |l^{-1}(t) \cap Y|$ for every $t \in T$. We use the notations defined for so-structures also for LSOs.

We introduce an important subclass of so-structures similar to the subclass of total orders in the case of partial orders.

Definition 2 (Total linear so-structure). *An so-structure $\mathcal{S} = (V, \prec, \sqsubset)$ is called* total linear *if* $\text{co}_{\prec} = (\sqsubset \setminus \prec) \cup id_V$. *The set of all total linear extensions (or* linearizations*) of an so-structure $\mathcal{S}'$ is denoted by $lin(\mathcal{S}')$.*

Total linear so-structures are maximally sequentialized in the sense that no further $\prec$- or $\sqsubset$- relations can be added maintaining the requirements of so-structures according to Definition 1. Therefore the linearizations $lin(\mathcal{S}')$ of an so-structure $\mathcal{S}'$ are its maximal extensions. Note that a total linear so-structure $lin = (V, \prec, \sqsubset)$ represents a sequence of (synchronous) steps $\tau_1 \ldots \tau_n$ (we also write $lin = \tau_1 \ldots \tau_n$). A (synchronous) step is a set of cyclic $\sqsubset$-ordered events (forming a so called $\sqsubset$-clique – such events can only occur synchronously as explained in the context of Figure 1) and the sequential ordering is caused by $\prec$-relations between these steps. That means $\tau_1 \ldots \tau_n$ and $(V, \prec, \sqsubset)$ are related through $V = \bigcup_{i=1}^{n} \tau_i$, $\prec = \bigcup_{i<j} \tau_i \times \tau_j$ and $\sqsubset = ((\bigcup_{i=1}^{n} \tau_i \times \tau_i) \setminus id_V) \cup \prec$. For example, the linearizations of the run $\kappa(AON)$ in Figure 1 are the sequences of (synchronous) steps $tu\{v, w\}$ and $\{t, u\}\{v, w\}$. By abstracting from the nodes of a total linear LSO $lin = (V, \prec, \sqsubset, l)$ representing $\tau_1 \ldots \tau_n$, every step (set) of events τ_i can be interpreted as a step (multi-set) $l(\tau_i)$ of transitions using the labeling function. This is a general principle. That means we will interpret such a (synchronous) step sequence $\tau_1 \ldots \tau$ of events based on a total linear LSO $lin = (V, \prec, \sqsubset, l)$ as a sequence $\sigma_{lin} = l(\tau_1) \ldots l(\tau_n)$ of (synchronous) transition steps in a Petri net. Thus, we often do not distinguish total linear LSOs and respective sequences of transition steps in a Petri net. Lastly we need the notion of prefixes of so-structures. These are defined by subsets of nodes which are downward closed w.r.t. the $\sqsubset$-relation:

Definition 3 (Prefix). *Let $\mathcal{S} = (V, \prec, \sqsubset)$ be an so-structure and let $V' \subseteq V$ be a set of events such that $u' \in V'$, $u \sqsubset u' \Longrightarrow u \in V'$. Then V' is called* prefix *w.r.t. $\mathcal{S}$. A* prefix V' of $u \in V \setminus V'$ *is a prefix w.r.t. $\mathcal{S}$ satisfying $(v \prec u \Longrightarrow v \in V')$.*

The prefixes w.r.t. $\kappa(AON)$ in Figure 1 are the event sets $\{t\}$, $\{t, u\}$ and $\{t, u, v, e\}$. The only prefix of w is $\{t, u\}$, since v and w may not occur in a prefix of w ($w \sqsubset v$) and u has to occur in a prefix of w ($u \prec w$). We have the following relation between prefixes and linearizations of so-structures:

Lemma 1. *Let V' be a prefix (of $u \in V$) w.r.t. an so-structure $\mathcal{S} = (V, \prec, \sqsubset)$, then there exists $lin \in lin(\mathcal{S})$ such that V' is a prefix (of u) w.r.t. lin.*

Proof. $lin = \tau_1 \dots \tau_n$ can be constructed as follows: $\tau_1 = \{v \in V' \mid \forall v' \in V' : v' \not\prec v\}$, $\tau_2 = \{v \in V' \setminus \tau_1 \mid \forall v' \in V' \setminus \tau_1 : v' \not\prec v\}$ and so on, i.e. we define $\tau_i \subseteq V'$ as the set of nodes $\{v \in V' \setminus (\bigcup_{j=1}^{i-1} \tau_j) \mid \forall v' \in V' \setminus (\bigcup_{j=1}^{i-1} \tau_j) : v' \not\prec v\}$ which are minimal w.r.t. the restriction of $\prec$ onto the node set $V' \setminus (\bigcup_{j=1}^{i-1} \tau_j)$, as long as $V' \setminus (\bigcup_{j=1}^{i-1} \tau_j) \neq \emptyset$. Then continue with the same procedure on $V \setminus V' = V \setminus (\bigcup_{j=1}^{i} \tau_j)$, i.e. $\tau_{i+1} = \{v \in V \setminus (\bigcup_{j=1}^{i} \tau_j) \mid \forall v' \in V \setminus (\bigcup_{j=1}^{i} \tau_j) : v' \not\prec v\}$ and so on. By construction V' is a prefix (of u) w.r.t. lin. A straightforward computation also yields $lin \in lin(\mathcal{S})$. $\square$

A prefix V' w.r.t. a total linear so-structure $lin = \tau_1 \dots \tau_n$ always represents a primary part of the respective (synchronous) step sequence, i.e. $V' = \bigcup_{j \leq i} \tau_j$ for some $i \in \{0, \dots, n\}$. If V' is a prefix of u, then $u \in \tau_{i+1}$.

Next we present the net class of pti-nets (p/t-nets with weighted inhibitor arcs). As usual, a *p/t-net* is a triple $N = (P, T, W)$, where P is a finite set of places, T is a finite set of transitions and $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the weight function representing the flow relation. The pre- and post-multi-set of a transition $t \in T$ are the multi-sets of places given by ${}^\bullet t(p) = W(p, t)$ and $t^\bullet\,(p) = W(t, p)$ for all $p \in P$. This notation can be extended to $U \in \mathbb{N}^T$ by ${}^\bullet U(p) = \sum_{t \in U} U(t)\, {}^\bullet t(p)$ and $U^\bullet\,(p) = \sum_{t \in U} U(t) t^\bullet\,(p)$ for all $p \in P$. Analogously we can define pre- and post-multi-sets of multi-sets of places as multi-sets of transitions. Each $m \in \mathbb{N}^P$ is called a *marking* of N and each $U \in \mathbb{N}^T$ is called a step of N. U is *enabled to occur* in m if and only if $m \geq {}^\bullet U$. In this case, its occurrence leads to the marking $m' = m - {}^\bullet U + U^\bullet$.

Definition 4 (Pti-net). *A marked* pti-net *is a quadruple $NI = (P, T, W, I, m_0)$, where* $\mathrm{Und}(NI) = (P, T, W)$ *is a p/t-net (the* underlying net *of NI), m_0 the* initial marking *of NI and $I : P \times T \to \mathbb{N} \cup \{\infty\}$ is the* inhibitor (weight) function *(we assume $\infty > n$ for every $n \in \mathbb{N}$). For a transition t the negative context ${}^-t \in (\mathbb{N} \cup \{\infty\})^P$ is given by ${}^-t(p) = I(p, t)$ for all $p \in P$. For a step of transitions U, ${}^-U \in (\mathbb{N} \cup \{\infty\})^P$ is given by ${}^-U(p) = min(\{\infty\} \cup \{{}^-t(p) \mid t \in U\})$. A place p with ${}^-t(p) \neq \infty$ is called* inhibitor place *of t.*

A step of transitions U is (synchronously) enabled to occur *in a marking m if and only if it is enabled to occur in the underlying p/t-net* $\mathrm{Und}(NI)$ *and in addition $m \leq {}^-U$. The occurrence of U leads to the marking $m' = m - {}^\bullet U + U^\bullet$. This is denoted by $m \xrightarrow{U} m'$. A finite sequence of steps of transitions $\sigma = U_1 \dots U_n$, $n \in \mathbb{N}$, is* *called a* step (occurrence) sequence enabled in a marking m and leading to m_n*, denoted by* $m \xrightarrow{\sigma} m_n$*, if there exists a sequence of markings* $m_1, \ldots, m_n$ *such that* $m \xrightarrow{U_1} m_1 \xrightarrow{U_2} \ldots \xrightarrow{U_n} m_n$*. By* $\mathcal{EX}(NI)$ *we denote the set of all step sequences of a marked pti-net* NI*.*

Note that $I(p,t) = k \in \mathbb{N}$ implies that t can only occur if p does not contain more than k tokens (as explained in the context of the inhibitor arc connected with w in Figure 1); $k = 0$ coincides with zero-testing. Accordingly $I(p,t) = \infty$ means that the occurrence of t is not restricted through the presence of tokens in p. Thus a p/t-net can always be interpreted as a pti-net with $I \equiv \infty$. In graphic illustrations, inhibitor arcs are drawn with circles as arrowheads and annotated with their weights (see Figure 1). Inhibitor arcs with weight ∞ are completely omitted and the inhibitor weight 0 is not shown in diagrams. The definition of enabledness in Definition 4 reflects the considerations about the a-priori testing explicated above: the inhibitor constraints are obeyed before the step of transitions is executed. For an example, see Figure 1 and the explanations at the beginning of this section.

Now we introduce the process semantics for pti-nets as presented in [13]. The problem is that the absence of tokens in a place – this is tested by inhibitor arcs – cannot be directly represented in an occurrence net. This is solved by introducing local extra conditions and read arcs – also called activator arcs – connected to these conditions. These extra conditions are introduced "on demand" to directly represent dependencies of events caused by the presence of an inhibitor arc in the net. The conditions are artificial conditions without a reference to inhibitor weights or places of the net. They only focus on the dependencies that result from inhibitor tests. Thus, activator arcs represent local information regarding the lack of tokens in a place. The process definition of [13] is based on the usual notion of occurrence nets extended by activator arcs. These are (labeled) acyclic nets with non-branching places (conditions) (since conflicts between transitions are resolved). By abstracting from the conditions one obtains an LSO representing the causal relationships between the events. In the following definition B represents the finite set of *conditions*, E the finite set of *events*, R the flow relation and Act the set of activator arcs of the occurrence net.

Definition 5 (Activator occurrence net). *A* labeled activator occurrence net *(ao-net) is a five-tuple* $\mathrm{AON} = (B, E, R, Act, l)$ *satisfying:*

- *B and E are finite disjoint sets,*
- *$R \subseteq (B \times E) \cup (E \times B)$ and $Act \subseteq B \times E$,*
- *$|{}^\bullet b|, |b^\bullet| \leq 1$ for every $b \in B$,*
- *the relational structure $\mathcal{S}(\mathrm{AON}) = (E, \prec_{loc}, \sqsubset_{loc}, l|_E) = (E, (R \circ R)|_{E \times E} \cup (R \circ Act), (Act^{-1} \circ R) \setminus id_E, l|_E)$ is $\diamondsuit$-acyclic,*
- *l is a labeling for $B \cup E$.*

The LSO generated by AON *is* $\kappa(\mathrm{AON}) = (E, \prec_{\mathrm{AON}}, \sqsubset_{\mathrm{AON}}, l|_E) = \mathcal{S}(\mathrm{AON})^{\diamondsuit}$.

The relations $\prec_{loc}$ and $\sqsubset_{loc}$ represent the local information about causal relationships between events. Figure 2 shows their construction rule. $\kappa(\mathrm{AON})$ captures all (not only local) causal relations between the events (see also Figure 1). Note that Definition 5 is a conservative extension of common occurrence nets by read arcs.

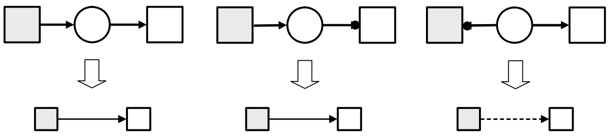


Fig. 2. Generation of the orders $\prec_{loc}$ and $\sqsubset_{loc}$ in *ao*-nets

The initial marking $\mathrm{MIN_{AON}}$ of AON consists of all conditions without incoming flow arcs (the minimal conditions w.r.t. R). The final marking $\mathrm{MAX_{AON}}$ of AON consists of all conditions without outgoing flow arcs (the maximal conditions w.r.t. R). There are two different notions of configurations and slices for *ao*-nets. A set of events $D \subseteq E$ is a *strong configuration* of AON, if $e \in D$ and $f \prec_{loc}^{+} e$ implies $f \in D$. D is called a *weak configuration* of AON, if $e \in D$ and $f(\prec_{loc} \cup \sqsubset_{loc})^{+}e$ implies $f \in D$. A *strong slice* of AON is a maximal (w.r.t. set inclusion) set of conditions $S \subseteq B$ which are incomparable w.r.t. the relation $R \circ \prec_{loc}^{*} \circ R$, denoted by $S \in \mathrm{SSL(AON)}$. A *weak slice* of AON is a maximal (w.r.t. set inclusion) set of conditions $S \subseteq B$ which are incomparable w.r.t. the relation $R \circ (\prec_{loc} \cup \sqsubset_{loc})^{*} \circ R$, denoted by $S \in \mathrm{WSL(AON)}$. In the example occurrence net from Figure 1 $|\mathrm{WSL}| = 4$ and $|\mathrm{SSL}| = 12$.

Every weak configuration is also a strong configuration and every weak slice is also a strong slice. In [13] it is shown that the set of strong slices of AON equals the set of all sets of conditions which are generated by firing the events of a strong configuration. An analogous result holds for weak slices and weak configurations. $\mathrm{SSL(AON)}$ equals the set of all sets of conditions reachable from the initial marking $\mathrm{MIN_{AON}}$ in AON and $\mathrm{WSL(AON)}$ equals the set of all sets of conditions from which the final marking $\mathrm{MAX_{AON}}$ is reachable in AON (using the standard a-priori occurrence rule of elementary nets with read arcs [13]). By $\mathrm{MAR}(C)$ we denote the marking resulting from the initial marking of a net by firing the multi-set of transitions corresponding to a (weak or strong) configuration C.

Now we are prepared to define processes of pti-nets as in [13]. The mentioned artificial conditions are labeled by the special symbol $\curlywedge$. They are introduced in situations, when a transition $t \in T$ tests a place in the pre- or post-multi-set of another transition $w \in T$ for absence of tokens, i.e. when $I(p,t) \neq \infty$ and ${}^{\bullet}w(p) + w^{\bullet}(p) \neq 0$ for some $p \in P$. Such situations are abbreviated by $w \multimap t$. If $w \multimap t$ holds, then any two occurrences f of w and e of t are adjacent to a common $\curlywedge$-condition representing a causal dependency of f and e. That means there exists a condition $b \in \widetilde{B}$ such that $(b,e) \in \mathit{Act}$ and ${}^{\bullet}f(b) + f^{\bullet}(b) \neq 0$ (remember that ${}^{\bullet}f, f^{\bullet} \in B^{\mathbb{N}}$ are multi-sets over B) – abbreviated by $f \multimap\!\!\bullet\, e$ (see requirement 6. in Definition 6). Thus the axiomatic process definition in [13] is as follows:

Definition 6 (Activator process). *An* activator process *(a-process) of NI is an ao-net* $\mathrm{AON} = (B \uplus \widetilde{B}, E, R, \mathit{Act}, l)$ *satisfying:*

1. *$l(B) \subseteq P$ and $l(E) \subseteq T$.*
2. *The conditions in $\widetilde{B} = \{b \mid \exists e \in E : (b,e) \in \mathit{Act}\}$ are labelled by the special symbol $\curlywedge$.*

3. $m_0 = l(\mathrm{MIN}_{\mathrm{AON}} \cap B)$.
4. *For all* $e \in E$, ${}^\bullet l(e) = l({}^\bullet e \cap B)$ *and* $l(e)^\bullet = l(e^\bullet \cap B)$.
5. *For all* $b \in \widetilde{B}$, *there are unique* $g, h \in E$ *such that* ${}^\bullet b + b^\bullet = \{g\}$, $(b, h) \in Act$ *and* $l(g) \multimap l(h)$.
6. *For all* $e, f \in E$, *if* $l(f) \multimap l(e)$ *then there is exactly one* $c \in \widetilde{B}$ *such that* $f \multimap\!\!\bullet\, e$ *through* c.
7. *For all* $e \in E$ *and* $S \in \mathrm{SSL}(\mathrm{AON})$, *if* ${}^\bullet e \cup \{b \in \widetilde{B} \mid (b, e) \in Act\} \subseteq S$ *then* $l(S \cap B) \leq {}^-l(e)$.

The set of a-processes of NI *(given by this axiomatic definition) is denoted by* $\alpha(NI)$. *For* $\mathrm{AON} \in \alpha(NI)$ *the generated so-structure* $\kappa(\mathrm{AON})$ *is called a run (associated to* AON*).*

The occurrence net AON in Figure 1 is indeed an a-process: All $\curlywedge$-labeled conditions satisfy 5. All $\curlywedge$-labeled conditions which are necessary according to 6. are drawn. Condition 7. must be simply verified for the strong slices produced by strong configurations, e.g. $\mathrm{MAR}(\emptyset)$, $\mathrm{MAR}(\{t\})$, $\mathrm{MAR}(\{u\})$, $\mathrm{MAR}(\{u, t\})$ and so on. Thus, $\kappa(\mathrm{AON})$ is a run.

The requirements 1., 3., 4. in Definition 6 represent common features of processes well-known from p/t-nets. They ensure that a-processes constitute a conservative generalization of common p/t-net processes. That means, the set of processes of $\mathrm{Und}(NI)$ coincides with the set of processes resulting from $\alpha(NI)$ by omitting the $\curlywedge$-labeled conditions (omitting the $\curlywedge$-conditions from an a-process AON leads to the so called underlying process UAON of AON). If NI has no inhibitor arcs (thus $NI = \mathrm{Und}(NI)$) a-processes coincide with common processes. Thus, Definition 6 can also be used to define processes of p/t-nets. The properties 2. and 5. together with the rule 6. – describing when $\curlywedge$-conditions have to be inserted – constitute the structure of the $\curlywedge$-conditions. The requirement 7. expresses that in the strong slices of AON the inhibitor constraints of the pti-net have to be properly reflected. That means, for events enabled in a certain slice of AON the respective transitions are also enabled in the respective marking in the pti-net NI.

We finally formally define, when we consider an LSO $\mathcal{S}$ to be consistent with the step semantics $\mathcal{EX}$ of a given pti-net (Definition 4). Such LSOs we call *enabled* (w.r.t. the given pti-net). Intuitively it is clear what enabledness means: The transitions associated to the events of an LSO can be executed in the net regarding all given concurrency and dependency relations. For the formal definition the concurrency and dependency relations described by $\mathcal{S}$ are reduced to the set of step sequences sequentializing $\mathcal{S}$ (given by $lin(\mathcal{S})$). Such step sequences can be considered as observations of $\mathcal{S}$, where transition occurrences within a step are observed at the same time (synchronously), and step occurrences are observed in the order given by the step sequence. If each such observation of $\mathcal{S}$ is an enabled step occurrence sequences of the pti-net, $\mathcal{S}$ is consistent with the step semantics.

Definition 7 (Enabled LSO). *An LSO* $\mathcal{S} = (V, \prec, \sqsubset, l)$ *is enabled w.r.t. a marked pti-net* $NI = (P, T, W, I, m_0)$ *if and only if every* $lin \in lin(\mathcal{S})$ *represents an enabled (synchronous) step sequence* σ_{lin} *in* $\mathcal{EX}(NI)$ *(of* NI*).* $\mathcal{ELCS}(NI)$ *is the set of all so-structures enabled w.r.t. a given marked pti-net* NI.

With this definition one can easily check that the run $\kappa(AON)$ in Figure 1 is enabled w.r.t. NI: The two linearizations of $\kappa(AON)$ represent the sequences of synchronous steps $tu\{v,w\}$ and $\{t,u\}\{v,w\}$ which are both executable in NI.

Definition 7 is consistent with and a proper generalization of the notion of enabled LPOs in the context of p/t-nets: An LPO $\mathrm{lpo} = (V, \prec, l)$ with $l : V \to T$ is *enabled w.r.t. a marked p/t-net* (P, T, W, m_0) if each step sequence which extends lpo is a step occurrence sequence enabled in m_0. Since in LPOs concurrent and synchronous transition occurrences are not distinguished, here a step is considered as a set of events labeled by transitions (transition occurrences) which are concurrent.

Beside the consistency of Definition 7 with the definition of enabled LPOs, there are two general semantical arguments justifying this definition: First the set of total linear LSOs $lin(\mathcal{S})$, which are tested for enabledness in the Petri net, represents $\mathcal{S}$. This is shown in [6] by the following generalization of Szpilrajns theorem [16] to so-structures: $\mathcal{S} = (V, \bigcap_{(V,\prec,\sqsubset)\in lin(\mathcal{S})} \prec, \bigcap_{(V,\prec,\sqsubset)\in lin(\mathcal{S})} \sqsubset)$. Second the set $lin(\mathcal{S})$ can express arbitrary concurrency relations between transition occurrences of a pti-net, since concurrency equals the possibility of sequential occurrence in any order and synchronous occurrence. Thus, considering more generally sequences of concurrent steps of synchronous steps instead of simply sequences of synchronous steps does not lead to a higher expressivity of concurrency. These two arguments justify the choice of synchronous step sequences as the operational semantics (of executions) of pti-nets. Thus the definition of enabled LSOs based on synchronous step sequences and total linear LSOs constitutes the adequate causal semantics.

3 The Semantical Framework

In [13] a general framework for dealing with process semantics of Petri nets was proposed (see Figure 3, left part). It aims at a support for a systematic development of process and causality semantics for various Petri net classes using a common scheme.

In Figure 3 the abbreviations mean the following. $\mathcal{PN}$ represents a Petri net model together with an operational occurrence rule. $\mathcal{EX}$ are executions such as step sequences in accordance to the occurrence rule employed by $\mathcal{PN}$. $\mathcal{LAN}$ represents the process semantics given by labeled acyclic nets such as occurrence nets. $\mathcal{LEX}$ are labeled executions such as step sequences of nets in $\mathcal{LAN}$. Finally, $\mathcal{LCS}$ are labeled causal structures describing net behavior through causality relations between events. The arrows indicate functions that define and relate the different semantical views. They represent the consistency requirements for process semantics according to this framework. ω yields the set of executions (step sequences) providing the operational semantics (Definition 4 for pti-nets). α defines the axiomatic process definition (Definition 6). κ associates so called runs to the process definition (Definition 6); $\kappa(\mathcal{LAN}) \subseteq \mathcal{LCS}$ defines the set of runs of a net. λ represents the operational semantics of the process definition given by labeled step sequences (defined through a slight modification of the step occurrence rule of elementary nets with read arcs under the a-priori semantics [13]). Through ϕ a labeled execution can be interpreted as an ordinary execution (defined as trivial modification omitting labels). ϵ and ι relate a labeled causal structure with its generated

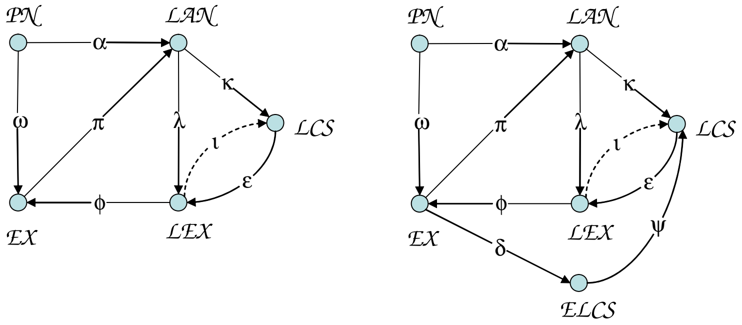


Fig. 3. Left: The semantical framework of [13]. Right: The left semantical framework extended by the completeness-requirement that any enabled causal structure has to be a sequentialization of a run; this is depicted through $\mathcal{ELCS}$ and the adjacent arcs labeled by δ and ψ

labeled executions (ϵ respectively ι are given as linearizations respectively intersections in the case of LSOs). Finally, π represents the operational process definition starting from executions.

This framework defines reasonable requirements for process semantics. It provides a schematic approach to ensure that process and causality semantics developed for a special Petri net class are consistently defined. In [13] the framework is condensed to five properties that have to be checked in each particular setting. Two of these properties state that all mappings in Figure 3 are total and all mappings returning sets do not return the empty set. *Consistency* is formulated there as the following separated properties:

Soundness: The process definition $\mathcal{LAN}$ should be *sound* w.r.t. the step semantics $\mathcal{EX}$ in the sense that every run should be consistent with the step semantics.
Weak completeness: $\mathcal{LAN}$ should be *weak complete* w.r.t. $\mathcal{EX}$ in the sense that $\mathcal{EX}$ should be reproducible from $\mathcal{LAN}$.
Construction of processes from step sequences: A process in $\mathcal{LAN}$ should be constructible from each step sequence in $\mathcal{EX}$ generated by the process (by π).
Consistency of runs and processes (called *Fitting* in [13])): Processes and corresponding runs should generate the same step sequences.
Runs are reconstructible from step sequences (called *Representation* in [13])): Runs from $\mathcal{LCS}$ should be reconstructible from step sequences in $\mathcal{EX}$ by $\iota \circ \epsilon$.

But an important feature of process semantics relating runs and step semantics is not present in this framework. On the one hand, $\phi \circ \epsilon$ ensures that each run is consistent with the step semantics (soundness). On the other hand, there is no requirement guaranteeing the converse, that each causal structure which is consistent with the step semantics is generated by a run through adding causality to it (completeness). For p/t-nets this is fulfilled (as mentioned in the Introduction), since every enabled LPO is a sequentialization of a run [11]. Together with the reverse statement that runs are enabled (soundness), completeness guarantees that there are runs and processes which express all valid causal behavior of the net regarding as much concurrency as possible. That means, the minimal causal dependencies in a net are reflected in the process semantics. To represent such an aim of completeness, we add new relations to the semantical framework (Figure 3, right part) by the introduction of enabled causal structures $\mathcal{ELCS}$. The arc labeled by δ represents the definition of enabled labeled causal structures $\mathcal{ELCS}$ from the operational semantics $\mathcal{EX}$. The arc labeled with ψ relates enabled labeled causal structures ($\mathcal{ELCS}$) and runs ($\kappa(\mathcal{LAN}) \subseteq \mathcal{LCS}$) in the above sense by assigning a run with less causality to each enabled labeled causal structure (for which such a run exists). Formally, a labeled causal structure is said to have *less causality* then a second one, if each labeled execution in $\mathcal{EX}$ generated by the second one is also generated by the first one (where the labeled executions generated by a labeled causal structure are given by ϵ). Thus, through $\psi \circ \delta$ we add an additional property to the process framework that we call the aim of completeness.

Definition 8 (Aim of completeness). *The mapping δ assigns a set of step sequences $\mathcal{EX}$ onto the set of causal structures $\mathcal{ELCS}$ enabled w.r.t. $\mathcal{EX}$. The mapping ψ assigns a run $\mathcal{LCS}$ with less causality to each enabled causal structure in $\mathcal{ELCS}$ for which such a run exists.*

The aim of completeness *states that the mapping ψ is total, i.e. that each enabled causal structure adds causality to some run.*

The absence of the aim of completeness in the framework of [13] leads to process definitions that do not have to represent minimal causal behavior. According to [13] a process definition that equals the operational step semantics (processes are step sequences) is a valid process semantics. But the set of step sequences is not a reasonable process semantics and process definitions not producing the minimal causalities are not really useful. The aim of completeness in our framework solves this problem. It implies that minimal enabled labeled causal structures coincide with (minimal) runs: On the one hand a minimal enabled labeled causal structure has to be a sequentializations of a run, on the other hand runs have to be enabled – so runs cannot have less causalities than minimal enabled labeled causal structures.

4 Process Semantics of Pti-nets

The definition of a-processes from section 2 meets all requirements of the left semantical framework in Figure 3 as shown in [13]. In the setting of pti-nets the additional aim of completeness states that each enabled so-structure extends some run of the pti-net. We show in this section that a-processes do not fulfill the aim of completeness. Moreover, we develop an alternative process definition preserving all the other requirements of the semantical framework, such that the aim of completeness is fulfilled.

The basic intuition behind the fact that the a-processes from Definition 6 do not generate minimal causalities is as follows: The definition uses constraints introduced through artificial $\curlywedge$-labeled conditions. They do not have counterparts on the pti-net level, but rather represent dynamic causal relationships between events. Therefore, it is possible that the definition of the $\curlywedge$-conditions does not reflect the causalities in the original pti-net such that too many constraints are introduced in the runs generated by

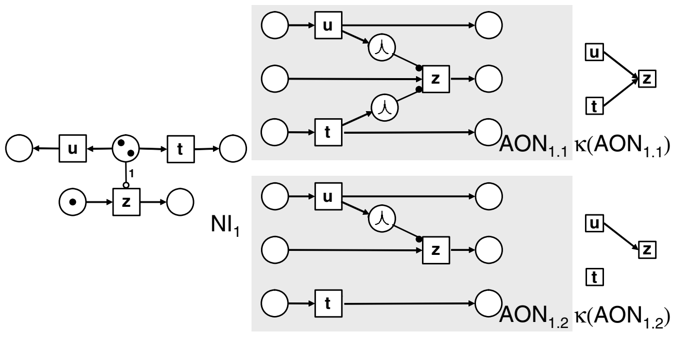


Fig. 4. A pti-net NI_1, an a-process $\mathrm{AON}_{1.1}$ of NI_1 and the associated run $\kappa(\mathrm{AON}_{1.1})$ together with an ao-net $\mathrm{AON}_{1.2}$ that is a candidate to be a process of NI_1, and the associated run $\kappa(\mathrm{AON}_{1.2})$. This example from [13] shows that a-processes (mandatory) introduce unnecessary causalities.

a-processes. In this section we will step by step illustrate via examples why the aim of completeness does not hold for a-processes and adapt their definition such that this aim is finally fulfilled (all the other requirements will be preserved).

In the following we give two examples of LSOs enabled w.r.t. a marked pti-net, which do not extend a run of the considered net. Each of these examples leads to a specific modification of Definition 6. We assume that events in these examples are labeled by the identity mapping, i.e. u, t and z are events representing the occurrence of the transitions $l(u) = u$, $l(t) = t$ and $l(z) = z$. The place connected to z by an inhibitor arc in each example we denote by p.

The first example gave the authors of [13] themselves. The a-process $\mathrm{AON}_{1.1}$ in Figure 4 shows that the technique of introducing $\curlywedge$-labeled conditions according to Definition 6 in general generates too many constraints in the associated run $\kappa(\mathrm{AON}_{1.1})$: "One may easily verify that we can safely delete one of the activator arcs (but not both), which leads to another a-process generating weaker constraints than $AON_{1.1}$". Indeed, deleting for example the $\curlywedge$-condition between t and z the resulting ao-net $\mathrm{AON}_{1.2}$ is a reasonable process. The other $\curlywedge$-condition orders u and z in sequence $u \rightarrow z$ and t can occur concurrently to this sequence. On the other hand, omitting the $\curlywedge$-condition between t and z contradicts 6. of Definition 6 because there holds $t \multimap z$. That means $\mathrm{AON}_{1.2}$ is not an a-process (in particular the quoted statement is not exactly true). Thus, the LSO $\kappa(\mathrm{AON}_{1.2})$ is enabled but does not sequentialize a run (since it can only be generated by an ao-net without a $\curlywedge$-condition adjacent to t and z). An analogous observations holds symmetrically when deleting the $\curlywedge$-condition between u and z instead between t and z. Consequently, the first modification of Definition 6 is to replace requirement 6. by requirement 6.'. According to 6.', the unique condition $c \in \widetilde{B}$ is only possible instead of required. Then the problem discussed above is solved and the ao-net $\mathrm{AON}_{1.2}$ is actually a process.

6.' For all $e, f \in E$, if $f \multimap\!\bullet\, e$ then there is exactly one $c \in \widetilde{B}$ such that $f \multimap\!\bullet\, e$ through c.

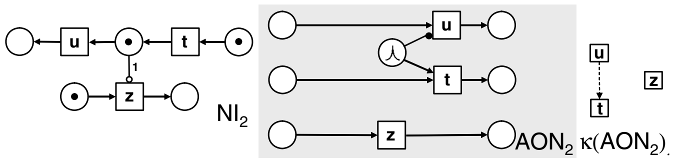


Fig. 5. A pti-net NI_2, an *ao*-net AON_2 that is a candidate to be a process of NI_2, and the associated run $\kappa(\mathrm{AON}_2)$. The *ao*-net models executable causalities that cannot be generated with a-processes.

The net NI_2 of Figure 5 shows that the aim of completeness is still not fulfilled: If u and t occur causally ordered in sequence $u \to t$ then z can fire concurrently to this sequence because the place p never contains more than one token. It is even possible to fire z concurrently to the synchronous step $\{u, t\}$. Consequently $\kappa(\mathrm{AON}_2)$, requiring solely that u occurs "not later than" t, is enabled (check Definition 7). The only possibility to introduce such a causal dependency between u and t on the process level is through a $\curlywedge$-condition between u and t. This is illustrated by the ao-net AON_2 (compare Figure 2). But according to 5. of Definition 6, AON_2 is not an a-process, since $l(u) \not\multimap l(t)$. Thus, a run which is extended by $\kappa(\mathrm{AON}_2)$ has no ordering between u, t and z. This is not possible because such a run is not enabled (the step sequence $t \to z \to u$ cannot be fired). That means $\kappa(\mathrm{AON}_2)$ does not sequentialize a run. Altogether, in 5. an important possibility of generating causal dependencies from inhibitor arcs via $\curlywedge$-conditions is not present. Allowing $\curlywedge$-conditions as in AON_2 solves this problem leading to a process having $\kappa(\mathrm{AON}_2)$ as its associated run. This $\curlywedge$-condition represents the causal dependency of u and t caused by the inhibitor arc (p, z). It reflects the inhibitor testing of z and not of u or t. A generalization of 5. allowing $\curlywedge$-conditions also in situations as in this example is a next necessary step towards the aim of completeness. Loosely speaking, we will allow to insert $\curlywedge$-conditions additionally in the following situation: If a transition, testing some place via an inhibitor arc, occurs concurrently to transitions consuming and producing tokens in this place, these transition occurrences must eventually be ordered via a $\curlywedge$-condition. This $\curlywedge$-conditions is intended to ensure that tokens are consumed not later than produced in order to restrict the maximal number of tokens in this place according to the inhibitor weight. To this end, we replaces 5. by the weaker requirement 5.'. It introduces a more general structural construction rule of $\curlywedge$-conditions using this intuition as follows:

5.' For all $b \in \widetilde{B}$, there are unique $g, h \in E$ such that ${}^\bullet b + b^\bullet = \{g\}$, $(b, h) \in Act$ and additionally $l(g) \multimap l(h)$ or ${}^\bullet l(h) \cap l(g)^\bullet \cap {}^-z \neq \emptyset$ for a $z \in T$.

But the modifications proposed so far still do not ensure that AON_2 is a process, since AON_2 does not fulfill 7. of Definition 6: The conditions resulting from only firing t in the initial marking establish a strong slice S and z fulfills ${}^\bullet z \cup \{b \in \widetilde{B} \mid (b, z) \in Act\} \subseteq S$. That means that using the standard occurrence rule of elementary nets with read arcs under the a-priori semantics [13] S constitutes a reachable marking in the process net and z is enabled in this marking in the process net. But obviously in the pti-net z is not enabled in the marking resulting from firing t. This problem can be resolved as follows: In AON_2 the event t can fire in the initial marking, although the $\curlywedge$-condition generates the ordering "u not later than t". Thus, firing t in the initial marking disables u. This means that we could have omitted u from AON_2 which leads to a different ao-net. Consequently, it is a proper assumption that ao-nets should model only such behavior in which every event of the ao-net actually occurs. Under this assumption, firing t in the initial marking is not a valid behavior of the ao-net and therefore the problematic marking S is not a marking of interest. The markings of interest are the markings reachable from the minimal conditions ($\mathrm{MIN}_{\mathrm{AON}_2}$) in the ao-net from which we can reach the maximal conditions ($\mathrm{MAX}_{\mathrm{AON}_2}$). That means, all events of the ao-net not fired yet can still be executed starting in the respective marking. These markings are represented by the weak slices of the ao-net. Therefore, we replace 7. by 7.', where SSL (strong slices) are replaced by WSL (weak slices) reflecting the above assumption:

7.' For all $e \in E$ and $S \in \mathrm{WSL}(\mathrm{AON})$, if ${}^\bullet e \cup \{b \in \widetilde{B} \mid (b,e) \in \mathit{Act}\} \subseteq S$ then $l(S \cap B) \leq {}^- l(e)$.

This is a generalization of Definition 6 since $\mathrm{WSL} \subseteq \mathrm{SSL}$. From the intuitive point of view the two alternative formulations 7. and 7.' focus on different aspects: While the consideration of SSL completely reflects the occurrence rule of elementary nets with read arcs, the consideration of WSL additionally postulates that no event of the ao-net may completely be disabled. This second assumption is also used in [13] for defining the executions $\mathcal{LEX}$ through the mapping λ in the semantical framework of Figure 3: λ represents all step sequences of an a-process in $\mathcal{LAN}$ in which every event of the process occurs. In this sense the change of the occurrence rule of ao-nets explained above is an adaption to the idea of mandatory regarding all events used in the operational semantics of ao-nets anyway. Therefore, this slightly altered occurrence rule of ao-nets (that we will use) is completely consistent to the executions of ao-nets and thus even fits better into the semantical framework.

Replacing 5., 6. and 7. by 5.', 6.' and 7.' in Definition 6 as described here ensures that the ao-net AON_2 is a process. So the above considerations lead to the following alternative process definition and thus a change of the mapping α in Figure 3 (denoted by α' instead of α in Definition 9):

Definition 9 (Complete activator process). *A* complete activator process *(ca-process) of NI is an ao-net* $\mathrm{AON} = (B \uplus \widetilde{B}, E, R, \mathit{Act}, l)$ *satisfying:*

1. *$l(B) \subseteq P$ and $l(E) \subseteq T$.*
2. *The conditions in $\widetilde{B} = \{b \mid \exists e \in E : (b,e) \in \mathit{Act}\}$ are labelled by the special symbol $\curlywedge$.*
3. *$m_0 = l(\mathrm{MIN}_{\mathrm{AON}} \cap B)$.*
4. *For all $e \in E$, ${}^\bullet l(e) = l({}^\bullet e \cap B)$ and $l(e)^\bullet = l(e^\bullet \cap B)$.*

5.' *For all $b \in \widetilde{B}$, there are unique $g, h \in E$ such that ${}^\bullet b + b^\bullet = \{g\}$, $(b,h) \in \mathit{Act}$ and additionally $l(g) \multimap l(h)$ or ${}^\bullet l(h) \cap l(g)^\bullet \cap {}^- z \neq \emptyset$ for a $z \in T$.*

6.' *For all $e, f \in E$, if $f \multimap\!\!\bullet\, e$ then there is exactly one $c \in \widetilde{B}$ such that $f \multimap\!\!\bullet\, e$ through c.*

7.' *For all $e \in E$ and $S \in \mathrm{WSL}(\mathrm{AON})$, if ${}^\bullet e \cup \{b \in \widetilde{B} \mid (b,e) \in \mathit{Act}\} \subseteq S$ then $l(S \cap B) \leq {}^- l(e)$.*

The set of ca-processes of NI *is denoted by* $\alpha'(NI)$. *For* $\mathrm{AON} \in \alpha'(NI)$ *the generated so-structure* $\kappa(\mathrm{AON})$ *is called a run (associated to* AON*).*

Note that the requirements 1.,3.,4. of Definition 6 are preserved in Definition 9 and thus also ca-processes constitute a conservative generalization of common p/t-net processes. Omitting the $\curlywedge$-conditions from a ca-process AON leads to the so called underlying process $\mathrm{Und}(\mathrm{AON})$ of AON, which is a process of $\mathrm{Und}(NI)$. We will show now as the main result of this paper that the *ca*-process definition actually fulfills the aim of completeness. Due to lack of space, we only give a sketch of the proof (which has three pages). The complete proof can be found in the technical report [10].

Theorem 1. *For every enabled LSO* $\mathcal{S} = (E, \prec, \sqsubset, l)$ *of a pti-net* NI *there exists a ca-process* $\mathrm{AON} \in \alpha'(NI)$ *whereas* $\mathcal{S}$ *is an extension of the run* $\kappa(\mathrm{AON})$.

Proof (Sketch). The LPO $\mathrm{lpo}_{\mathcal{S}} = (E, \prec, l)$ underlying $\mathcal{S}$ is enabled w.r.t. $\mathrm{Und}(NI)$. Thus there exists a process $\mathrm{UAON} = (B, E, R', l')$ of $\mathrm{Und}(NI)$ fulfilling that $\mathrm{lpo}_{\mathcal{S}}$ sequentializes the run $\kappa(\mathrm{UAON})$. The basic idea is now to construct an *ao*-net AON from UAON by adding all $\curlywedge$-conditions to UAON which can be added according to property 5.' while not producing causal dependencies contradicting $\mathcal{S}$. Then this *ao*-net $\mathrm{AON} = (B \uplus \widetilde{B}, E, R, Act, l)$ is the sought ca-process. It is clear that AON satisfies 1.-4., 5.' and 6.'. Thus, it only remains to show that AON meets condition 7.' of Definition 9, i.e. that given $e \in E$ and $S \in \mathrm{WSL}(\mathrm{AON})$ with ${}^{\bullet}e \cup \{b \in \widetilde{B} \mid (b, e) \in Act\} \subseteq S$ it holds that $l(S \cap B) \leq {}^{-}l(e)$. For this, we fix a weak configuration C of AON with $S = \mathrm{MAR}(C)$ and show that $l(e)$ is executable in the pti-net after the occurrence of the transitions corresponding to events in C. To this end, we define a prefix C_{pre} of e in $\mathcal{S}$ containing as many events from C as possible. Using that $\mathcal{S}$ is enabled, we can deduce that $l(e)$ is executable in the pti-net after the occurrence of the transitions corresponding to events in C_{pre}: By Lemma 1 there is $lin \in lin(\mathcal{S})$ such that C_{pre} is a prefix of e w.r.t. lin. Because $\mathcal{S}$ is enabled the total linear so-structure $lin = \tau_1 \ldots \tau_n$ represents an enabled synchronous step sequence of NI with $C_{pre} = \bigcup_{j=1}^{i-1} \tau_j$ and $e \in \tau_i$ (for $i \in \{1 \ldots n\}$). This implies that e can occur after C_{pre}. Finally C_{pre} can be transformed in several steps into the set C and in each step it can be shown that the transformation does not disable $l(e)$. □

In the following we briefly explain that the other aims of the semantical framework are still fulfilled by the new process definition:

Soundness: Using Proposition 5.19 of [13] it is easy to see that every run is enabled, i.e. if $\mathrm{AON} \in \alpha'(NI)$, then $\phi(\epsilon(\kappa(\mathrm{AON}))) \subseteq \omega(NI)$.
Consistency of runs and processes: Processes and runs generate the same step sequences, i.e. if $\mathrm{AON} \in \alpha'(NI)$, then $\epsilon(\kappa(\mathrm{AON})) = \lambda(\mathrm{AON})$ (that means the rules for constructing causal relationships between events from processes as shown in Figure 2 are correct). This follows since in proposition 5.19 of [13] this relation was shown for arbitrary *ao*-nets (note here that the construction rules of the involved mappings λ, κ and ϵ have not changed in contrast to [13], only the process definition constituting the starting point of this relation is changed).

Weak completeness: Any execution of the pti-net ($\mathcal{EX}$) given by $\omega(NI)$ is generated from a ca-process, i.e. for any execution $\sigma \in \mathcal{EX}$ there exists an ca-process AON $\in \alpha'(NI)$ with $\sigma \in \phi(\lambda(\text{AON}))$ ($\omega(NI) \subseteq \bigcup_{\text{AON}\in\alpha'(NI)} \phi(\lambda(\text{AON}))$). This also holds for ca-processes, because this is the relation generalized in comparison to a-processes (the aim of completeness is a generalization of the weak completeness property).
Runs are reconstructible from step sequences: Each run is the intersection of all observations it generates, i.e. $\iota \circ \epsilon$ reconstructs a run. This relation holds because of the generalization of Szpilrajns theorem to so-structures described in the preliminaries (note that in this context nothing is changed in contrast to [13]).
Construction of processes from step sequences: There is no obvious way to generalize the constructive definition of π from [13] because especially the new requirement 6.' of Definition 9 is problematic: Now it is no more mandatory but optional to introduce $\curlywedge$-conditions between certain transitions (the transition candidates can be identified with 5.') and one has to check whether 7.' holds (7. holds by construction). There is the following constructive process definition that is based directly on the axiomatic definition: Given an enabled step sequence σ of NI a ca-processes can be generated as follows:

- Construct a usual p/t-net process of $Und(NI)$ (based on an occurrence net) starting from σ.
- Introduce arbitrary $\curlywedge$-labeled conditions in accordance with 5.' and 6.' of Definition 9.
- Check 7.' of Definition 9: if it is fulfilled the construction is finished, else perform the next step.
- Introduce further $\curlywedge$-labeled conditions in accordance with 5.' and 6.' of Definition 9, then go back to the previous step.

All processes constructible with this algorithm produce the set of ca-processes $\pi'(\sigma)$ generated by σ. Moreover, the ca-processes generated from a step sequence σ are the ca-processes having σ (provided with respective labels) as an execution. This algorithm always terminates because there are only finite many possible $\curlywedge$-labeled conditions in accordance with 5.' and 6.' of Definition 9. Introducing *all* such possible $\curlywedge$-conditions obviously leads to a ca-process, i.e. 7.' is then fulfilled in step 3. More precisely, the number of possible $\curlywedge$-conditions is at most quadratic in the number of events which means that the number of repetitions of the steps 3 and 4 of the algorithm is polynomial. Thus, only checking 7.' in step 3 may be not efficient, since there exists an exponential number of (weak) slices in the number of nodes. But current research results on a similar topic summarized in [14] show that there exists an algorithm polynomial in time solving this problem: In [14] we present an algorithm (based on flow theory) that can be used to calculate step 3 in polynomial time (of degree $O(n^3)$). Therefore, with this construction the requirements interrelated with the mapping π in the semantical framework of Figure 3 are also fulfilled.

5 Conclusion

In this paper we have developed a general semantical framework that supports the definition of process semantics and respective causal semantics for arbitrary Petri net classes.

The framework is based on the semantical framework from [13] additionally requiring that process semantics should be complete w.r.t. step semantics: Each causal structure which is consistent to step semantics – such causal structures we call enabled – should be generated from a process net. Since for the description of causal net behavior of pti-nets under the a-priori semantics labeled so-structures are applied, the notion of enabled so-structures has been introduced. We were able to show that the process definition for pti-nets from [13] is not complete w.r.t. step semantics and to identify a structural generalization of this process definition which is complete (while still satisfying all the other requirements of the framework of [13]).

Possible further applications of the results of this paper are on the one hand the usage of the semantical framework on further Petri net classes in order to check existing process semantics and to evolve new process semantics. In the context of the paper, this is in particular interesting for existing inhibitor net semantics [19,6,2,12,13,8]: While most aims of [13] are checked for those process semantics, the new aim of completeness is not (probably because this is the most complicated aim). Nevertheless a lot of these process semantics seem to satisfy the aim of completeness (at least for the process semantics of elementary nets with inhibitor arcs under the a-priori semantics as well as the a-posteriori semantics there are formal proofs [9]). On the other hand the ca-processes of this paper constitute a process definition for pti-nets under the a-priori semantics expressing minimal causalities and can thus be useful e.g. for model checking algorithms based on unfoldings.

References

1. Billington, J.: Protocol specification using p-graphs, a technique based on coloured petri nets. In: Reisig, W., Rozenberg, G. [20] pp. 293–330
2. Busi, N., Pinna, G.M.: Process semantics for place/transition nets with inhibitor and read arcs. Fundam. Inform. 40(2-3), 165–197 (1999)
3. Donatelli, S., Franceschinis, G.: Modelling and analysis of distributed software using gspns. In: Reisig, W., Rozenberg, G. [20], pp. 438–476
4. Goltz, U., Reisig, W.: The non-sequential behaviour of petri nets. Information and Control 57(2/3), 125–147 (1983)
5. Goltz, U., Reisig, W.: Processes of place/transition-nets. In: Díaz, J. (ed.) Automata, Languages and Programming. LNCS, vol. 154, pp. 264–277. Springer, Heidelberg (1983)
6. Janicki, R., Koutny, M.: Semantics of inhibitor nets. Inf. Comput. 123(1), 1–16 (1995)
7. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. In: Monographs in Theoretical Computer Science, vol. 1-3, Springer, Heidelberg (1992) (1994) (1997)
8. Juhas, G.: Are these events independend? it depends! Habilitation (2005)
9. Juhas, G., Lorenz, R., Mauser, S.: Synchronous + concurrent + sequential = earlier than + not later than. In: Proceedings of ACSD 2006, pp. 261–270 (2006)
10. Juhás, G., Lorenz, R., Mauser, S.: Complete process semantics of inhibitor net (2007) Technical report http://www.informatik.ku-eichstaett.de/mitarbeiter/lorenz/techreports/complete.pdf
11. Kiehn, A.: On the interrelation between synchronized and non-synchronized behaviour of petri nets. Elektronische Informationsverarbeitung und Kybernetik 24(1/2), 3–18 (1988)

12. Kleijn, H.C.M., Koutny, M.: Process semantics of p/t-nets with inhibitor arcs. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 261–281. Springer, Heidelberg (2000)
13. Kleijn, H.C.M., Koutny, M.: Process semantics of general inhibitor nets. Inf. Comput. 190(1), 18–69 (2004)
14. Lorenz, R., Bergenthum, R., Mauser, S.: Testing the executability of scenarios in general inhibitor nets. In: Proceedings ACSD 2007 (2007)
15. Peterson, J.: Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs (1981)
16. Szpilrajn, E.: Sur l'extension de l'ordre partiel. Fundamenta Mathematicae 16, 386–389 (1930)
17. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)
18. Vogler, W.: Partial words versus processes: a short comparison. In: Rozenberg, G. (ed.) Advances in Petri Nets: The DEMON Project. LNCS, vol. 609, pp. 292–303. Springer, Heidelberg (1992)
19. Vogler, W.: Partial order semantics and read arcs. In: Privara, I., Ruzicka, P. (eds.): MFCS 1997. LNCS, vol. 1295, pp. 508–517. Springer, Heidelberg (1997)
20. Reisig, W., Rozenberg, G. (eds.): Lectures on Petri Nets II: Applications, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996. LNCS, vol. 1492. Springer, Heidelberg (1998)

Behaviour-Preserving Transition Insertions in Unfolding Prefixes

Victor Khomenko

School of Computing Science, Newcastle University
Newcastle upon Tyne, NE1 7RU, U.K.
Victor.Khomenko@ncl.ac.uk

Abstract. Some design methods based on Petri nets modify the original specification by behaviour-preserving insertion of new transitions. If the unfolding prefix is used to analyse the net, it has to be re-unfolded after each modification, which is detrimental for the overall performance.

The approach presented in this paper applies the transformations directly to the unfolding prefix, thus avoiding re-unfolding. This also helps in visualisation, since the application of a transformation directly to the prefix changes it in a way that was 'intuitively expected' by the user, while re-unfolding can dramatically change the shape of the prefix. Moreover, rigourous validity checks for several kinds of transition insertions are developed. These checks are performed *on the original unfolding prefix,* so one never has to backtrack due to the choice of a transformation which does not preserve the behaviour.

Keywords: Petri net unfoldings, transition insertions, transformations, Petri nets, encoding conflicts, STGs, asynchronous circuits.

1 Introduction

Some design methods based on Petri nets modify the original specification by behaviour-preserving insertion of new transitions. For example, Signal Transition Graphs (STGs) are a formalism widely used for describing the behaviour of asynchronous control circuits. Typically, they are used as a specification language for the synthesis of such circuits [2,5,18]. STGs are a class of interpreted Petri nets, in which transitions are labelled with the names of rising and falling edges of circuit signals. In the discussion below, though we have in mind a particular application, viz. synthesis of asynchronous circuits from STG specifications, almost all the developed techniques and algorithms are not specific to this application domain and suitable for general Petri nets (e.g., one can envisage the applications to action refinement).

Circuit synthesis based on STGs involves: (i) checking the necessary and sufficient conditions for the STG's implementability as a logic circuit; (ii) modifying, if necessary, the initial STG to make it implementable; and (iii) finding an appropriate Boolean cover for the next-state function of each output and internal signals, and obtaining them in the form of Boolean equations for the logic gates of the circuit.

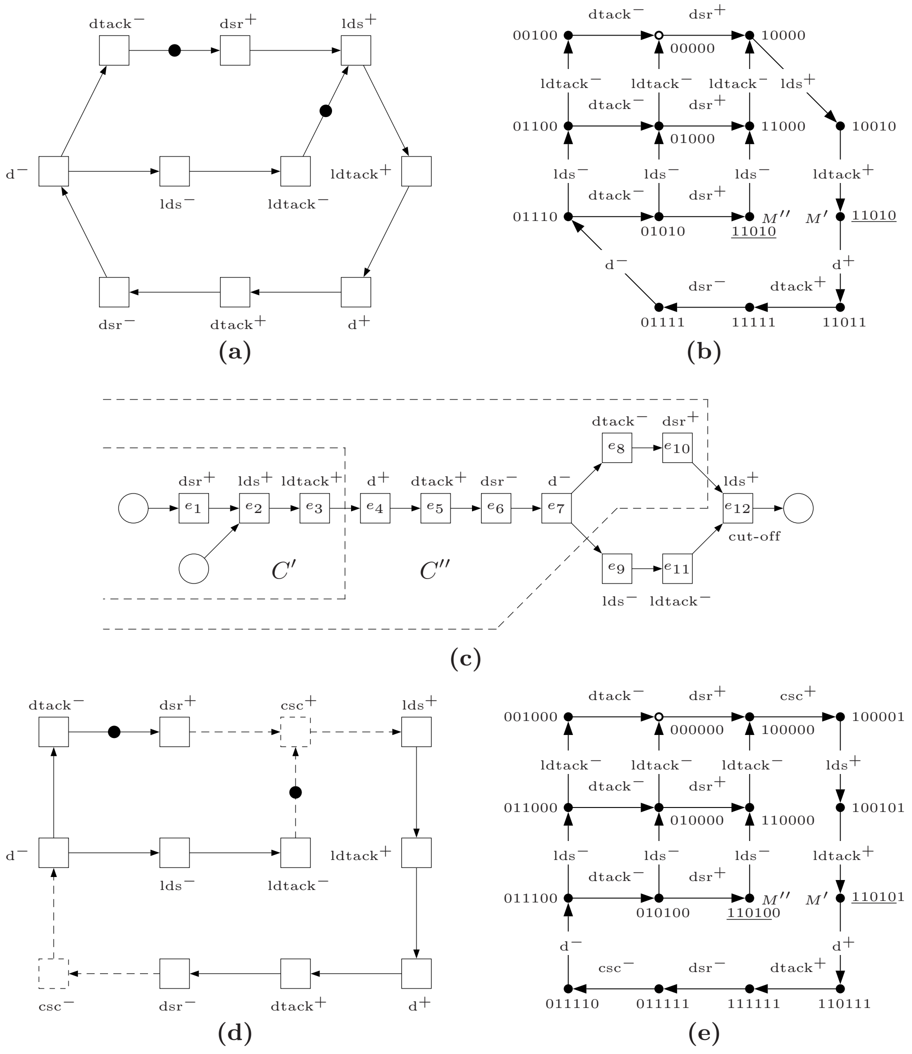
Step (i) of this process may detect *state encoding conflicts*, which occur when semantically different reachable markings (i.e., enabling different sets of output signals) of the STG have the same binary *encoding*, i.e., the binary vector containing the value of each signal in the circuit, as illustrated in Figures 1(a,b). A specification containing such encoding conflicts is not directly implementable: intuitively, at the implementation level the only information available to the circuit is the encoding, and so it is unable to distinguish between the conflicting states.

To proceed with the synthesis, one first has to *resolve* the encoding conflicts (step (ii) of the process), which is usually done by adding one or more new *internal* signals helping to distinguish between conflicting states, as illustrated in Figures 1(d,e). Hence, the original STG has to be modified by insertion of new transitions, in such a way that its 'external' behaviour does not change. Intuitively, insertion of new signals extends the encoding vector, introducing thus additional 'memory' helping the circuit to trace the current state.

One of the commonly used STG-based synthesis tools, Petrify [3,5], performs all of these steps automatically, after first constructing the state graph (in the form of a BDD [1]) of the initial STG specification. While the state graph based approach is relatively well-studied, the issue of computational complexity for highly concurrent STGs is quite serious due to the *state space explosion* problem [17]; that is, even a relatively small STG can (and often does) yield a very large state space. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive, especially if the STG models are not constructed manually by a designer but rather generated automatically from high-level hardware descriptions (e.g., Petrify often fails to synthesise circuits with more than 20–25 signals).

In order to alleviate this problem, Petri net analysis techniques based on causal partial order semantics, in the form of Petri net unfoldings [6,8], were applied to circuit synthesis. Since in practice STGs usually exhibit a lot of concurrency, but have rather few choice points, their complete unfolding prefixes are often exponentially smaller than the corresponding state graphs; in fact, in many of the experiments conducted in [8,12] they are just slightly bigger then the original STGs themselves. Therefore, unfolding prefixes are well-suited for both visualisation of an STG's behaviour and alleviating the state space explosion problem. The papers [12,13,14] present a complete design flow for complex-gate logic synthesis based on Petri net unfoldings, which completely avoids generating the state graph, and hence has significant advantage both in memory consumption and in runtime, without affecting the quality of the solutions. Moreover, unfoldings are much more visual than state graphs (the latter are hard to understand due to their large sizes and the tendency to obscure causal relationships and concurrency between the events), which enhances the interaction with the user.

Arguably, the most difficult task in the complex-gate logic synthesis from STGs is resolution of encoding conflicts, which is usually done by signal insertion. This is the only part of the design flow presented in [12,13,14] which may



inputs: *dsr*, *ldtack*; **outputs:** *dtack*, *lds*, *d*; **internal:** *csc*

Fig. 1. An STG modelling a simplified VME bus controller **(a)**, its state graph with an encoding conflict between two states **(b)**, a finite and complete unfolding prefix with two configurations corresponding to the CSC conflict **(c)**, a modified STG where the encoding conflict has been resolved by adding a new signal *csc* **(d)**, and its state graph **(e)**. The order of signals in the binary encodings is: *dsr*, *ldtack*, *dtack*, *lds*, *d*[, *csc*].

require human intervention. In fact, the techniques presented in [14] are targeted at facilitating the interaction with the user, by developing a method for visualisation of encoding conflicts.

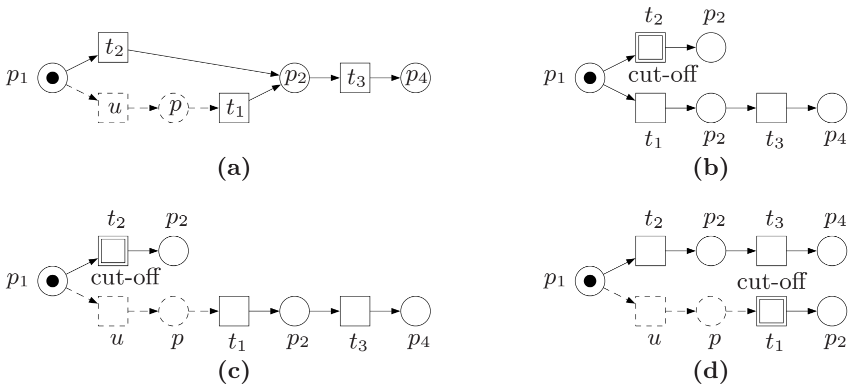


Fig. 2. A Petri net with the transformation shown in dashed lines **(a)**, an unfolding prefix before the transformation **(b)**, the 'intuitively expected' unfolding prefix of the modified net **(c)**, and the result of re-unfolding **(d)**. The unfolding algorithm proposed in [6,8] was used for **(b,d)**.

The tool described in [14] works as follows. First, the STG is unfolded and the encoding conflicts are computed and visualised. Then a set of potentially useful signal insertions is computed and arranged according to a certain cost function. Then the user selects one of these transformations, the STG is modified and the process is repeated until all the encoding conflicts are eliminated. It is currently the responsibility of the user to ensure that the selected transformation is valid — although some validity checks are performed by the tool, it does not guarantee the correctness. Moreover, some of these correctness checks are performed *after* the STG has been modified and re-unfolded, i.e., the tool has to backtrack if the chosen transformation happens to be incorrect (e.g., due to the user's mistake).

The approach presented in this paper improves that in [14] in several ways. First, it applies the transformation not only to the STG, but also directly to the unfolding prefix, thus avoiding re-unfolding at each step of the method. This also helps in visualisation, since the application of the transformation directly to the prefix changes it in a way that was 'intuitively expected' by the designer, while re-unfolding of the modified STG can dramatically change the shape of the prefix (due to different events being declared cut-off, as illustrated in Figure 2) to which the designer got 'used to'. Moreover, rigourous checks of correctness are developed. These checks are performed *on the original unfolding prefix,* so the algorithm never has to backtrack due to the choice of an incorrect transformation. These features also make the described approach easier for full automation, as described in [10].

Also, there are some problem-specific advantages. In general, not all the encoding conflicts are resolved by a single transformation (hence the need for multiple iterations in the approach of [14]). If the shape of the prefix has changed only slightly and in a predictable way, the unresolved encoding conflicts computed for the original prefix can be transferred to the modified one, which is not generally possible with re-unfolding. This considerably improves the efficiency of the method.

It should be noted that performing the transformations directly on the prefix is not trivial, since one has to guarantee the completeness of the resulting prefix (in fact, as shown below, naïve algorithms are incorrect). The main difficulty comes from the need to look *beyond cut-off events* of the prefix. Though the idea of transforming the prefix is not new, to our knowledge, this is the first time it is done with a rigourous proof of correctness. In fact, since the transformed prefix can be quite different from that obtained by re-unfolding the modified STG, it is complete in a different sense, and to justify the proposed approach we employ rather heavy machinery from the unfolding theory, viz. *canonical prefixes* [11]. Since the formal presentation requires from the reader familiarity with those techniques, it is delegated (together with all the proofs) to technical report [9] (available on-line), and this paper is aimed at an informal presentation of the results.

2 Basic Notions

In this section, we first present basic definitions concerning Petri nets, and then recall notions related to unfolding prefixes (see also [6,8,11,15]).

2.1 Petri Nets

A *net* is a triple $N \overset{\text{df}}{=} (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. A *marking* of N is a multiset M of places, i.e., $M : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. We adopt the standard rules about drawing nets, viz. places are represented as circles, transitions as boxes, the flow relation by arcs, and markings are shown by placing tokens within circles. In addition, the following short-hand notation is used: a transition can be connected directly to another transition if the place 'in the middle of the arc' has exactly one incoming and one outgoing arc (see, e.g., Figures 1(a,c,d)). As usual, ${}^\bullet z \overset{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \overset{\text{df}}{=} \{y \mid (z, y) \in F\}$ denote the *pre*- and *postset* of $z \in P \cup T$, and we define ${}^\bullet Z \overset{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \overset{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. We will assume that ${}^\bullet t \neq \emptyset$, for every $t \in T$. N is *finite* if $P \cup T$ is finite, and *infinite* otherwise. A *net system* or *Petri net* is a tuple $\Sigma \overset{\text{df}}{=} (P_\Sigma, T_\Sigma, F_\Sigma, M_\Sigma)$ where $(P_\Sigma, T_\Sigma, F_\Sigma)$ is a finite net and M_Σ is an *initial* marking. Whenever a new Petri net Σ is introduced, the corresponding elements P_Σ, T_Σ, F_Σ and M_Σ are also (implicitly) introduced.

We assume the reader is familiar with the standard notions of the theory of Petri nets (see, e.g., [15]), such as *enabling* and *firing* of a transition, marking *reachability*, *deadlock*, and net *boundedness* and *safeness*. A finite or infinite sequence $\sigma = t_1 t_2 t_3 \ldots$ of transitions is an *execution from a marking* M if t_1 can fire from M leading to a marking M' and $\sigma' = t_2 t_3 \ldots$ is an execution from M' (an empty sequence of transitions is an execution from any marking). Moreover, σ is an *execution of* Σ if it is an execution from M_Σ. For a transition $t \in T_\Sigma$ and a finite execution σ we will denote by $\#_t \sigma$ the number of occurrences of t in σ. A transition is *dead* if no reachable marking enables it, and *live* if from any reachable marking M there is an execution containing it. (Note that being live is a stronger property than being non-dead.)

2.2 Unfolding Prefixes

A *finite and complete unfolding prefix* Pref_Σ of a Petri net Σ is a finite acyclic labelled net which implicitly represents all the reachable states of Σ together with transitions enabled at those states. Intuitively, it can be obtained through unfolding Σ, by successive firings of transitions, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The resulting object Unf_Σ is called the *unfolding* of Σ. We will denote by h_Σ the function mapping the events and conditions of Unf_Σ to the corresponding places and transitions of Σ, and if $h_\Sigma(x) = y$ then we will refer to x as being *y-labelled* or as an *instance of y*. Unf_Σ is acyclic, and the precedence relation $\prec$ on its nodes will be called the *causal order*.

A *configuration* C is a finite set of events of Unf_Σ such that (i) for every $e \in C$, $f \prec e$ implies $f \in C$ (i.e., C is causally closed), and (ii) for all distinct $e, f \in C$, ${}^\bullet e \cap {}^\bullet f = \emptyset$ (i.e., there are no choices between the events of C). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of some of its events (viz. concurrent ones) is not important. For a transition t of Σ and a configuration C of Unf_Σ we will denote by $\#_t C$ the number of t-labelled events in C, and for an event e of Unf_Σ, $[e]_\Sigma$ will denote the *local* configuration of e, i.e., the minimal (w.r.t. $\subset$) configuration of Unf_Σ containing e. Moreover, $\mathit{Mark}(C)$ will denote the *final marking of C*, i.e., the marking of Σ reached by the execution $h_\Sigma(e_1)h_\Sigma(e_2)\ldots h_\Sigma(e_k)$, where $e_1, e_2, \ldots, e_k$ is any total ordering of the events of C consistent with $\prec$.

Unf_Σ is infinite whenever Σ has an infinite execution; however, if Σ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events beyond which it is not generated), yielding a finite prefix Pref_Σ. Unfolding algorithms declare an event e cut-off if there is a smaller (w.r.t. some well-founded partial order $\lhd$, called an *adequate order*, see [6,11]) *corresponding configuration* C in the already built part of the prefix containing no cut-off events and such that $\mathit{Mark}([e]) = \mathit{Mark}(C)$. It turns out that prefixes built in this way are *complete*, i.e., (i) every reachable marking M of Σ is represented in such a prefix by means of a configuration C containing no cut-off events and such that $\mathit{Mark}(C) = M$; and (ii) all the firings are preserved, i.e., if a configuration C of Pref_Σ containing no cut-off events can be extended by an event e of Unf_Σ then e is in Pref_Σ (it may be a cut-off event). Hence, the unfolding is truncated *without loss of information* and can, in principle, be re-constructed from Pref_Σ. For example, a finite and complete prefix of the STG in Figure 1(a) is shown in part (c) of this figure.

Efficient algorithms exist for building finite and complete prefixes [6,8], which ensure that the number of non-cut-off events in the resulting prefix never exceeds the number of reachable states of Σ. In fact, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. The experimental results in [12] demonstrate that high levels of compression can indeed be achieved in practice.

3 Transformations

In this paper, we are primarily interested in *SB-preserving* transformations, i.e., transformations preserving safeness and behaviour (in the sense that the original and the transformed Petri nets are weakly bisimilar, provided that the newly inserted transitions are considered silent) of the Petri net. This section describes several kinds of transition insertions.

3.1 Sequential Pre-insertion

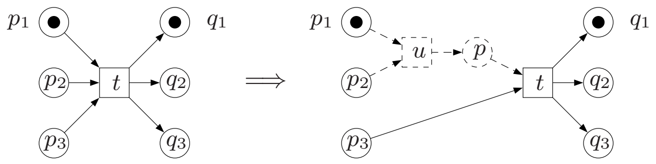
A sequential pre-insertion is essentially a generalised transition splitting, and is formally defined as follows.

Definition 1 (Sequential pre-insertion). *Given a Petri net Σ, a transition $t \in T_\Sigma$ and a non-empty set of places $S \subseteq {}^\bullet t$, the* sequential pre-insertion $S \wr t$ *is the transformation yielding the Petri net Σ^u, where*

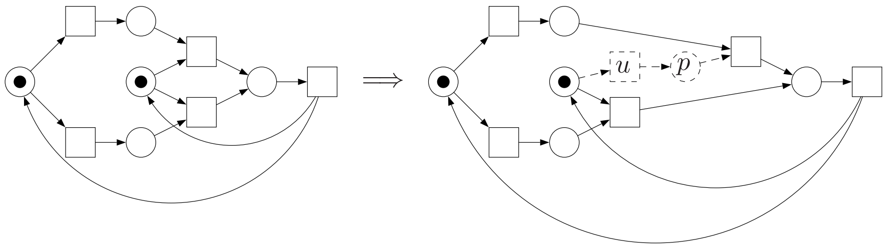
- $P_{\Sigma^u} \overset{\text{df}}{=} P_\Sigma \cup \{p\}$, *where $p \notin P_\Sigma \cup T_\Sigma$ is a new place;*
- $T_{\Sigma^u} \overset{\text{df}}{=} T_\Sigma \cup \{u\}$, *where $u \notin P_\Sigma \cup T_\Sigma \cup \{p\}$ is a new transition;*
- $F_{\Sigma^u} \overset{\text{df}}{=} (F_\Sigma \setminus \{(s,t) | s \in S\}) \cup \{(s,u) | s \in S\} \cup \{(u,p),(p,t)\}$;
- $M_{\Sigma^u}(q) \overset{\text{df}}{=} M_\Sigma(q)$ *for all $q \in P_\Sigma$, and $M_{\Sigma^u}(p) \overset{\text{df}}{=} 0$.*

We will write $\wr t$ instead of $S \wr t$ if $S = {}^\bullet t$, and $s \wr t$ instead of $\{s\} \wr t$. $\Diamond$

The picture below illustrates the sequential pre-insertion $\{p_1, p_2\} \wr t$.



One can easily show that sequential pre-insertion always preserves safeness and traces (traces are firing sequences with the silent (i.e., newly inserted) transitions removed). However, in general, the behaviour is not preserved, and so a sequential pre-insertion is not guaranteed to be SB-preserving. In fact, it can introduce deadlocks, as illustrated in the picture below.



Hence, one has to impose additional restrictions on the transformation to guarantee that it is SB-preserving. One can easily show that it is enough to require that the newly inserted transition never 'steals' tokens from the preset of any enabled transition, i.e., its firing cannot disable any other transition (see [9, Proposition 1]). This condition is a simple reachability property which can be efficiently tested on the original unfolding prefix *before the transformation*: one has to check that for each transition $t' \in S^\bullet \setminus \{t\}$ there is no reachable marking M covering $S \cup {}^\bullet t'$. This test is co-NP-complete in the size of the prefix, but in practice the set S is small, and it can be efficiently performed using, e.g., the techniques described in [8]. Moreover, in important special cases, e.g., if $|S| = 1$ or $S = {}^\bullet t$, simple polynomial algorithms exist, and in the case $S^\bullet = \{t\}$, the property is always satisfied and so the reachability analysis can be skipped altogether.

3.2 Sequential Post-insertion

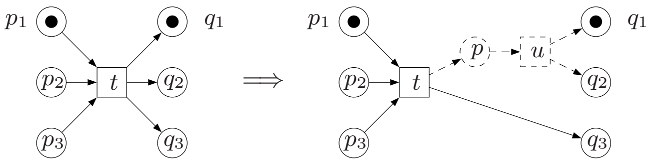
Similarly to sequential pre-insertion, sequential post-insertion is also a generalisation of transition splitting, and is formally defined as follows.

Definition 2 (Sequential post-insertion). *Given a Petri net Σ, a transition $t \in T_\Sigma$ and a non-empty set of places $S \subseteq t^\bullet$, the* sequential post-insertion $t \wr S$ *is the transformation yielding the Petri net Σ^u, where*

- $P_{\Sigma^u} \overset{\text{df}}{=} P_\Sigma \cup \{p\}$*, where $p \notin P_\Sigma \cup T_\Sigma$ is a new place;*
- $T_{\Sigma^u} \overset{\text{df}}{=} T_\Sigma \cup \{u\}$*, where $u \notin P_\Sigma \cup T_\Sigma \cup \{p\}$ is a new transition;*
- $F_{\Sigma^u} \overset{\text{df}}{=} (F_\Sigma \setminus \{(t,s) | s \in S\}) \cup \{(t,p),(p,u)\} \cup \{(u,s) | s \in S\}$*;*
- $M_{\Sigma^u}(q) \overset{\text{df}}{=} M_\Sigma(q)$ *for all $q \in P_\Sigma$, and* $M_{\Sigma^u}(p) \overset{\text{df}}{=} 0$.

We will write $t\wr$ instead of $t \wr S$ if $S = t^\bullet$, and $t \wr s$ instead of $t \wr \{s\}$. $\diamondsuit$

The picture below illustrates the sequential post-insertion $t \wr \{q_1, q_2\}$.



One can easily show that sequential post-insertions always preserve safeness and behaviour, and hence are always SB-preserving.

3.3 Concurrent Insertion

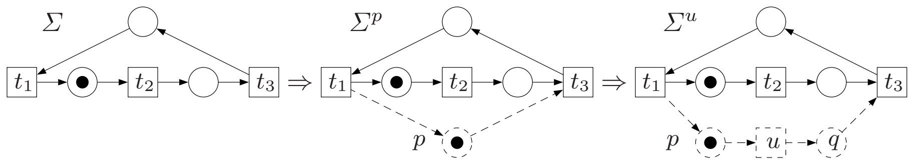
Concurrent transition insertion can be advantageous for performance, since the inserted transition can fire in parallel with the existing ones. It is defined as follows.

Definition 3 (Concurrent insertion). *Given a Petri net Σ, two of its transitions $t', t'' \in T_\Sigma$ and an $n \in \mathbb{N}$, the* concurrent insertion $t' \xrightarrow{\parallel n} t''$ *is the transformation yielding the Petri net Σ^u, where*

- $P_{\Sigma^u} \overset{\mathrm{df}}{=} P_\Sigma \cup \{p, q\}$, *where $p, q \notin P_\Sigma \cup T_\Sigma$ are two new places;*
- $T_{\Sigma^u} \overset{\mathrm{df}}{=} T_\Sigma \cup \{u\}$, *where $u \notin P_\Sigma \cup T_\Sigma \cup \{p, q\}$ is a new transition;*
- $F_{\Sigma^u} \overset{\mathrm{df}}{=} F_\Sigma \cup \{(t', p), (p, u), (u, q), (q, t'')\}$;
- $M_{\Sigma^u}(s) \overset{\mathrm{df}}{=} M_\Sigma(s)$ *for all $s \in P_\Sigma$, $M_{\Sigma^u}(p) \overset{\mathrm{df}}{=} n$ and $M_{\Sigma^u}(q) \overset{\mathrm{df}}{=} 0$.*

We will write $t' \xrightarrow{\parallel} t''$ instead of $t' \xrightarrow{\parallel 0} t''$ and $t' \xrightarrow{\parallel \bullet} t''$ instead of $t' \xrightarrow{\parallel 1} t''$. $\Diamond$

A concurrent insertion can be viewed as a two-stage transformation. In the first stage, a new place p with n tokens in it is inserted between t' and t''; this transformation will be denoted $t' \xrightarrow{\textcircled{n}} t''$ (or $t' \xrightarrow{\bigcirc} t''$ or $t' \xrightarrow{\odot} t''$ if n is 0 or 1, respectively), and the resulting Petri net will be denoted Σ^p. Then, the sequential pre-insertion $p \wr t''$ is applied. The picture below illustrates the concurrent insertion $t_1 \xrightarrow{\parallel \bullet} t_3$ (note that the token in p is needed to prevent a deadlock).



In general, concurrent insertions preserve neither safeness nor behaviour. In fact, safeness is not preserved even if $n = 0$ (e.g., when in the original net t' can fire twice without t'' firing), and deadlocks can be introduced even if $n = 1$ (e.g., when in the original net t'' should fire twice before t' can become enabled). Hence, one has to impose additional restrictions on the transformation to guarantee that it is SB-preserving.

Since $({}^\bullet u)^\bullet = \{u\}$, u cannot 'steal' a token from the preset of any other enabled transition, and thus $p \wr t''$ is always SB-preserving. Hence, instead of investigating the validity of a concurrent insertion $t' \xrightarrow{\parallel n} t''$, it is enough to investigate the validity of the corresponding place insertion $t' \xrightarrow{\textcircled{n}} t''$. One can observe that if the place inserted by the transformation $t' \xrightarrow{\textcircled{n}} t''$ is *implicit*[1] then the behaviour is preserved. Hence, checking that a place insertion $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving amounts to checking that the newly inserted place is safe and implicit in the resulting Petri net Σ^p; however, *these conditions should be checked using* $Pref_\Sigma$ *rather than* $Pref_{\Sigma^p}$.

Given a place insertion $t' \xrightarrow{\textcircled{n}} t''$ and a finite execution σ of Σ (respectively, a configuration C of Unf_Σ), we define $Tokens(\sigma) \overset{\mathrm{df}}{=} n + \#_{t'}\sigma - \#_{t''}\sigma$ (respectively, $Tokens(C) \overset{\mathrm{df}}{=} n + \#_{t'}C - \#_{t''}C$). Intuitively, $Tokens(\sigma)$ is the final number of tokens in the newly inserted place (provided that σ is an execution of the modified Petri net as well), i.e., this is the marking equation (see [15,16]) for this place. One can observe (see [9, Proposition 2]) that $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving iff for any finite execution σ of Σ, $Tokens(\sigma) \in \{0, 1\}$, or, equivalently, for any configuration C of Unf_Σ, $Tokens(C) \in \{0, 1\}$; i.e., t' and t'' should alternate in any execution σ of Σ, and, if $n = 0$ then t' should precede t'' in σ and if $n = 1$ then t'' should precede t' in σ.[2] Any execution of Σ or configuration of Unf_Σ violating this condition will be called *bad*.

One can show (see [9, Corollary 1]) that if $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving then either t' and t'' are dead or

$$n = \begin{cases} 1 & \text{if } \#_{t'}[e]_\Sigma = 0 \text{ for some } t''\text{-labelled event } e \text{ in } Pref_\Sigma \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

In effect, this means that in an SB-preserving place insertion $t' \xrightarrow{\textcircled{n}} t''$, only t' and t'' need to be specified, and n can be calculated using (1). Note that even if t' and t'' are dead, (1) still can be used to calculate n, since the choice of n does not matter in such a case.

Now we show how the correctness conditions formulated above can be checked using $Pref_\Sigma$. The main difficulty is that a bad configuration of Unf_Σ can contain cut-off events, and so a part of it can be not in $Pref_\Sigma$, i.e., one has to look *beyond cut-off events of the prefix.*

The key idea of the algorithm below is to check for each cut-off event e with a corresponding configuration C (note that $Mark([e]_\Sigma) = Mark(C)$) that after insertion of p the final markings of $[e]_{\Sigma^p}$ and C will still be equal, i.e., C will still be a corresponding configuration of e. This amounts to checking that $Tokens([e]_\Sigma) = Tokens(C)$. It turns out that if this condition holds and there is a bad configuration in Unf_Σ then one can find a bad configuration already in $Pref_\Sigma$ [9, Proposition 3].

The following algorithm, given t' and t'', checks whether the transformation $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving, where $n \in \{0, 1\}$ is computed using formula (1). The computation is performed using $Pref_\Sigma$ (no need to unfold the modified net).

[1] A place p is called *implicit* if the absence of tokens in it can never be the sole reason of any transition in $p^\bullet$ being disabled, i.e., if for each reachable marking M such that $M(p) = 0$ and for each transition $t \in p^\bullet$, $M(p') = 0$ for some place $p' \in {}^\bullet t \setminus \{p\}$, see [16].

[2] This property is closely related to the concept of *synchronic distance* [15].

Algorithm 1 (Checking correctness of a place insertion)

Inputs *$Pref_\Sigma$ and a place insertion $t' \xrightarrow{ⓟ} t''$ in Σ.*

Step 1 *If $Tokens([e]_\Sigma) \notin \{0,1\}$ for some instance e of t' or t'' in $Pref_\Sigma$ then reject the transformation and terminate.*
Step 2 *If $Tokens([e]_\Sigma) \neq Tokens(C)$ for some cut-off event e of $Pref_\Sigma$ with a corresponding configuration C then reject the transformation and terminate.*
Step 3 *Accept the transformation.*

One can show (see [9, Proposition 3]) that Algorithm 1 never accepts a non-SB-preserving transformation.[3] However, sometimes it can reject an SB-preserving transformation at Step 2. Nevertheless, this is conservative, and [9, Proposition 5] shows that if t' or t'' is live (i.e., in practically important cases) then Algorithm 1 is exact. Moreover, this algorithm runs in polynomial (in the size of $Pref_\Sigma$) time.

4 Insertions in the Prefix

Unfolding algorithms [6,8] compute at each step the set of *possible extensions*, i.e., the set of events which can be appended to the currently built part of the prefix. Since even the simpler problem of checking if this set is non-empty is NP-complete in the size of the prefix [7, Section 4.4], and the unfolding algorithms perform many such steps, their runtime can be quite large.

This section explains how to perform a transition insertion directly in the prefix, avoiding thus a potentially expensive re-unfolding. The main technical difficulty is that the resulting prefix can be very different from the one obtained by re-unfolding Σ^u, as illustrated in Figure 2. Thus it is not trivial to prove the completeness of the former prefix. For this, we obtain a characterisation of this prefix using a different adequate order, and apply the theory of canonical prefixes developed in [11] (the details can be found in the technical report [9]).

First, we establish the relationship between the configurations of Unf_Σ and Unf_{Σ^u}, assuming that Σ^u is obtained from Σ by a sequential pre- or post-insertion of a transition u. Below we denote by $\oplus$ the operation of extending a configuration by an event: $C \oplus e \overset{\mathrm{df}}{=} C \cup \{e\}$, provided that C and $C \cup \{e\}$ are configurations and $e \notin C$. Let C be a configuration of Unf_Σ and C^u be a configuration of Unf_{Σ^u}. It turns out that:

1. The set $\psi(C^u) \overset{\mathrm{df}}{=} \{e \in C^u \mid h_{\Sigma^u}(e) \neq u\}$ is a configuration of Unf_Σ.
2. There exists a unique configuration $\underline{\varphi}(C)$ of Unf_{Σ^u} containing no causally maximal instances of u and such that $\psi(\underline{\varphi}(C)) = C$. Moreover, there are

[3] In general, a cut-off event e can have multiple corresponding configurations. However, in practice only one of them is stored with the cut-off event. Hence one can imagine a situation when a cut-off event has several corresponding configurations and the property $Tokens([e]_\Sigma) = Tokens(C)$ holds for some of them but not the others. One can observe that Algorithm 1 rejects a non-SB-preserving transformation no matter which of these configurations was stored with e.

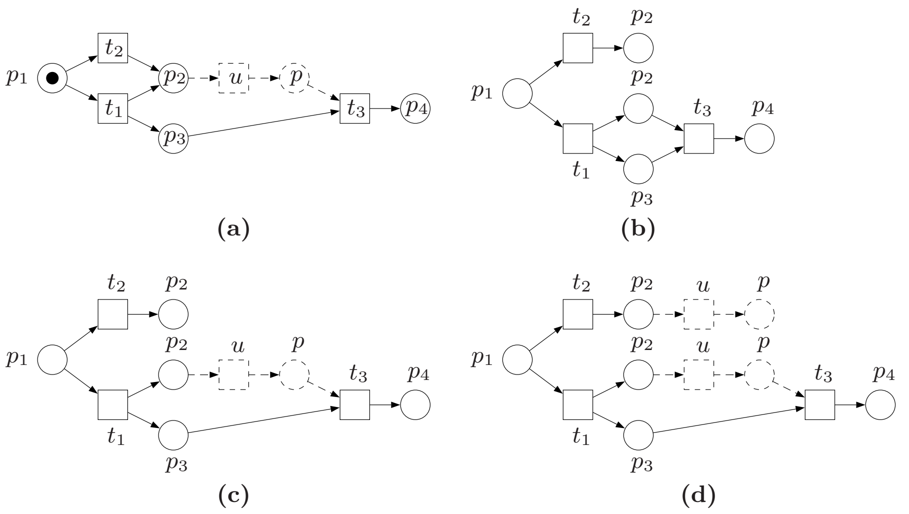


Fig. 3. A Petri net with the transformation shown in dashed lines **(a)**, its unfolding before the transformation **(b)**, the incomplete unfolding prefix obtained by naïve splitting **(c)**, and a complete unfolding prefix after the transformation **(d)**

at most two configurations in Unf_{Σ^u}, $\underline{\varphi}(C)$ and $\underline{\varphi}(C) \oplus e$ (where $h_{\Sigma^u}(e) = u$), such that $\psi(\underline{\varphi}(C)) = \psi(\underline{\varphi}(C \oplus e)) = C$. We define by $\overline{\varphi}(C)$ the latter configuration if it exists, and $\overline{\varphi}(C) \overset{\mathrm{df}}{=} \underline{\varphi}(C)$ otherwise.

Now, assuming that $\lhd$ is an adequate order on the configurations of Unf_{Σ}, we define the relation $\lhd^u$ on the configurations of Unf_{Σ^u} as $C' \lhd^u C''$ iff either $\psi(C') \lhd \psi(C'')$ or $\psi(C') = \psi(C'')$ and $\#_u C' < \#_u C''$. It turns out that $\lhd^u$ is an adequate order on the configurations of Unf_{Σ^u} [9, Proposition 7], and if one runs the unfolding algorithm for Σ^u using $\lhd^u$ as the adequate order and with some other minor changes discussed in [9, Propositions 8, 9] then the resulting prefix will coincide with that obtained by modifying $Pref_{\Sigma}$ using one of the algorithms discussed below.

4.1 Sequential Pre-insertion

Given a sequential pre-insertion $S \wr t$, we now show how to build $Pref_{\Sigma^u}$ from $Pref_{\Sigma}$. (Note that $S \wr t$ is not necessarily SB-preserving.) First of all, it should be noted that the naïve algorithm which simply splits each t-labelled event is, in general, incorrect: it can result in an incomplete prefix or even in an object which is not an unfolding prefix, as illustrated in Figures 3 and 4. Below we describe an algorithm based on a different idea. It inserts an instance of u in every position in the prefix where it is possible (much like a step of the unfolding algorithm) and then 're-wires' the instances of t.

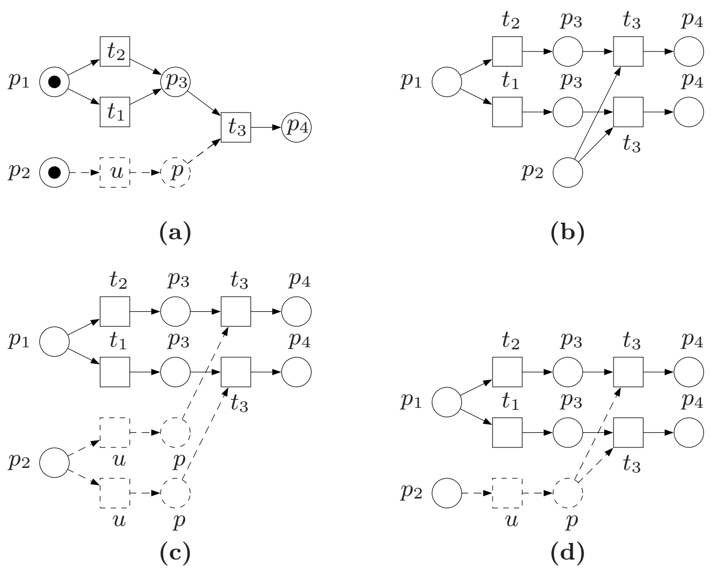


Fig. 4. A Petri net with the transformation shown in dashed lines **(a)**, its unfolding before the transformation **(b)**, the result of naïve splitting which is not an unfolding prefix due to the redundancy of nodes **(c)**, and the correct unfolding after the transformation **(d)**

Algorithm 2 (Sequential pre-insertion in the prefix)

Inputs *$Pref_\Sigma$ and a sequential pre-insertion $S \wr t$ in Σ.*
Outputs *A complete prefix of Σ^u.*

Step 1 *For each co-set*[4] *X containing no post-cut-off conditions and such that $h_\Sigma(X) = S$, create an instance of the new transition u, and make X its preset; create also an instance of the new place p, and make it the postset of the inserted transition instance.*
Step 2 *For each t-labelled event e (including cut-off events), let $X \subseteq {}^\bullet e$ be such that $h_\Sigma(X) = S$ (note that X is a co-set); moreover, let f be the (unique) u-labelled event with the preset X, and c be the p-labelled condition in $f^\bullet$. Remove the conditions in X from the preset of e, and add c there instead.*
Step 3 *For each cut-off event e with a corresponding configuration C, replace the corresponding configuration of e by $\underline{\varphi}(C)$.*

It is shown in [9, Proposition 8] that Algorithm 2 yields a correct prefix even if the pre-insertion is not SB-preserving (some additional information about the form of the resulting prefix is also given there).

[4] A set X of conditions of $Pref_\Sigma$ is a *co-set* if the conditions in X can be simultaneously marked, i.e., if there is a configuration C of $Pref_\Sigma$ such that $X \subseteq (\min_\prec Pref_\Sigma \cup C^\bullet) \setminus {}^\bullet C$, where $\min_\prec Pref_\Sigma$ is the set of causally minimal conditions of $Pref_\Sigma$.

In the worst case, the performance of this algorithm can be quite poor, since there can be an exponential in the size of the prefix number of co-sets X such that $h_\Sigma(X) = S$, and even a simpler problem of checking if there exists such a co-set is NP-complete. However, this algorithm still favourably compares to re-unfolding, since it is very similar to a *single step* of the unfolding algorithm. Moreover, in important special cases, e.g., if $|S| = 1$ or $S = {}^\bullet t$, this algorithm can be implemented to run in polynomial time.

4.2 Sequential Post-insertion

Given a sequential post-insertion $t \wr S$, we now show how to build $Pref_{\Sigma^u}$ from $Pref_\Sigma$. (Recall that sequential post-insertions are always SB-preserving, and so there is no need to check the validity of $t \wr S$.) The algorithm presented below is based on splitting u-labelled events, but special care should be taken when handling cut-off events (a naïve approach may result in an incomplete prefix, as illustrated in Figure 5). In particular, if a corresponding configuration C of a cut-off event e has an instance e' of t as a maximal event then e is not split (just its postset is amended), and the corresponding configuration becomes $\underline{\varphi}(C)$ (i.e., the instance of u after e' is not included into it).

Unfortunately, it may be no longer possible to choose the corresponding configurations of some of the cut-off events. In general, it is difficult to guarantee completeness without re-unfolding parts of the prefix, and the algorithm below can sometimes terminate unsuccessfully. In such a case, one either can re-unfold the Petri net (and thus the algorithm can be seen as a relatively cheap test whether a potentially much more expensive re-unfolding can be avoided) or simply discard the transformation (which makes sense when there are many alternative transformations to choose from).

Below, a configuration C of $Pref_\Sigma$ is called *u-extendible* if there is a t-labelled event $g \in C$ such that no instance $c \in g^\bullet$ of a place from S is in the preset of any event of C. (Intuitively, if C is u-extendible then the configuration $\underline{\varphi}(C)$ of $Pref_{\Sigma^u}$ can be extended by an instance of u).

Algorithm 3 (Sequential post-insertion in the prefix)

Inputs *$Pref_\Sigma$ and a sequential post-insertion $t \wr S$ in Σ.*
Outputs *A complete prefix of Σ^u.*

Step 1 *If there is a cut-off event e with a corresponding configuration C such that $[e]_\Sigma$ is u-extendible and C is not u-extendible then terminate unsuccessfully.*

Step 2 *For each t-labelled event e (including cut-off events): let $X \subseteq e^\bullet$ be the (unique) co-set satisfying $h_\Sigma(X) = S$. In the postset of e replace the conditions in X by a new instance c of p. If e is not a cut-off event then create a new instance of u with the preset $\{c\}$ and the postset X.*

Step 3 *For each cut-off event e of $Pref_\Sigma$ with a corresponding configuration C: replace the corresponding configuration of e by $\underline{\varphi}(C)$ if $[e]_\Sigma$ is u-extendible and by $\overline{\varphi}(C)$ otherwise. (In the latter case the corresponding configuration may become non-local, even if C was local).*

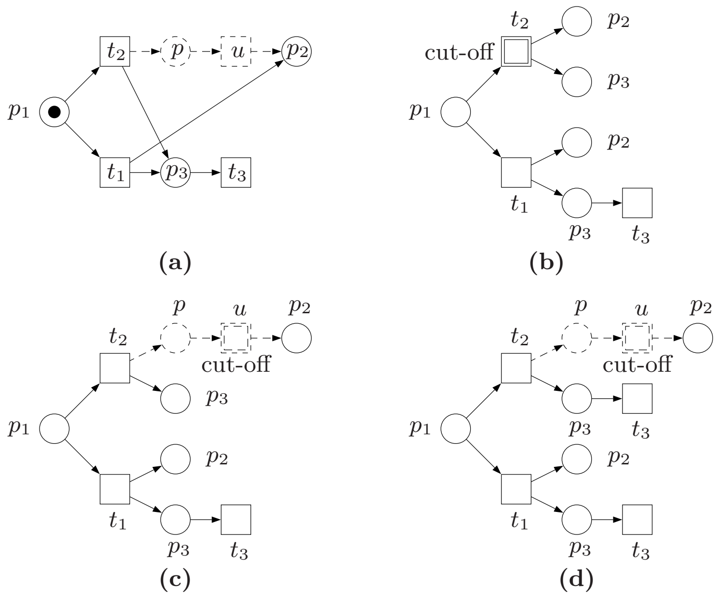


Fig. 5. A Petri net with a sequential post-insertion shown by dashed lines **(a)**, its unfolding before the transformation **(b)**, the incomplete unfolding prefix obtained as the result of naïve splitting **(c)**, and a complete unfolding prefix after the transformation **(d)**

It is shown in [9, Proposition 9] that if Algorithm 3 successfully terminates then the result is correct (some additional information about the form of the resulting prefix is also given there). Moreover, it runs in polynomial (in the size of $Pref_{\Sigma}$) time.

4.3 Concurrent Insertion

For clarity of presentation, the concurrent insertion $t' \xrightarrow{\|n} t''$ in the prefix is performed in two stages: first, a place insertion $t' \xrightarrow{ⓝ} t''$ is done, followed by the sequential pre-insertion $p \wr t''$, as explained in Section 3.3. (In practice, these two stages can easily be combined.) Furthermore, we assume that Algorithm 1 accepts the transformation $t' \xrightarrow{ⓝ} t''$. The following algorithm, given such a place insertion, builds $Pref_{\Sigma^p}$ from $Pref_{\Sigma}$. Intuitively, $Pref_{\Sigma^p}$ is obtained by adding a few p-labelled conditions to $Pref_{\Sigma}$, and appropriately connecting them to instances of t' and t''.

Algorithm 4 (Place insertion in the prefix)

Inputs *$Pref_{\Sigma}$ and a place insertion $t' \xrightarrow{ⓝ} t''$ in Σ accepted by Algorithm 1.*
Outputs *A complete prefix of Σ^p.*

Step 1 *If $n = 1$ then create a new p-labelled (causally minimal) condition.*
Step 2 *For each t'-labelled event e (including cut-off events), create a new p-labelled condition c and the arc (e, c).*
Step 3 *For each t''-labelled event e (including cut-off events): If $\#_{t'}[e]_{\Sigma} = 0$ then create a new arc (c, e), where c is the causally minimal p-labelled condition created in Step 1; else create a new arc (c, e), where c is the (unique) p-labelled condition in the postset of the (unique) causally maximal t'-labelled predecessor of e.*

It is shown in [9, Proposition 10] that if Algorithm 1 accepts a place insertion $t' \xrightarrow{ⓝ} t''$ then Algorithm 4 is correct (some additional information about the form of the resulting prefix is also given there). This follows from the fact that Algorithm 4 introduces no new causal constraints, since the instances of t'' consume only the conditions produced by their causal predecessors (or the one created in Step 1 of the algorithm), and so $Pref_{\Sigma^p}$ has the same set of configurations as $Pref_{\Sigma}$. Thus the adequate order does not change, and there is no need for the algorithm to amend the corresponding configurations of cut-off events.

Algorithm 4 runs in polynomial (in the size of $Pref_{\Sigma}$) time. Moreover, the algorithm performing a concurrent insertion in the prefix (composed of Algorithm 4 followed by Algorithm 2) is also polynomial, since the pre-insertion $p \wr t''$ is a special case for which Algorithm 2 can be implemented to run in polynomial time.

5 Optimisation

This section discusses several techniques allowing one to reduce the number of transformations which have to be considered, as well as to propagate information across different iterations of the algorithm for resolving encoding conflicts, avoiding thus repeating the same validity checks.

5.1 Equivalent Transformations

Sometimes a sequential post-insertion $t \wr S$ yields essentially the same net as a sequential pre-insertion $S' \wr t'$, where $t \in {}^{\bullet\bullet}t'$; in particular, this happens if $S \cup S' \subseteq t^{\bullet} \cap {}^{\bullet}t'$ and $|{}^{\bullet}p| = |p^{\bullet}| = 1$ for all $p \in S \cup S'$. In such a case there is no reason to distinguish between these two transformations, e.g., one can convert the post-insertion into an equivalent pre-insertion whenever possible. Moreover, since post-insertions are always SB-preserving, there is no need to check the validity of the resulting transformation.

5.2 Commutative Transformations

Two transformations *commute* if the result of their application does not depend on the order they are applied. (Note that a transformation can become ill-defined after applying another transformation, e.g., $t \wr \{p, q\}$ becomes ill-defined after applying $t \wr p$.) One can observe that:

- a concurrent insertion always commutes with any other transition insertion;
- a sequential pre-insertion and a sequential post-insertion always commute;
- two sequential pre-insertions $S \wr t$ and $S' \wr t'$ commute iff $t \neq t'$ or $S \cap S' = \emptyset$;
- two sequential post-insertions $t \wr S$ and $t' \wr S'$ commute iff $t \neq t'$ or $S \cap S' = \emptyset$.

It is important to note that an SB-preserving transition insertion remains SB-preserving if another commuting SB-preserving transition insertion is applied first. Hence transformations whose validity has been checked can be cached, and after some transformation has been applied, the non-commuting transformations are removed from the cache and the new transformations that became possible in the modified Petri net are computed, checked for validity and added to the cache. (In particular, in our application domain, there is no need to check the validity of a particular transformation if it was checked in some preceding iteration of the algorithm for resolving encoding conflicts.)

A *composite* transition insertion is a transformation defined as the composition of a set of pairwise commutative transition insertions. Clearly, if a composite transition insertion consists of SB-preserving transition insertions then it is SB-preserving, i.e., one can freely combine SB-preserving transition insertions, as long as they are pairwise commutative. This property comes useful for our application domain [10]: typically, several transitions of a new internal signal have to be inserted on each iteration of the algorithm for resolving encoding conflicts, in order to preserve the *consistency* [2,5] of the STG, i.e., the property that for every signal s, the following two conditions hold: (i) in all executions of the STG, the first occurrence of a transition of s has the same sign (either rising of falling); (ii) the rising and falling transitions of s alternate in every execution. (Consistency is a necessary condition for implementability of an STG as a circuit.) For example, in Figure 1(d) a composite transformation comprising two commuting SB-preserving sequential insertions (adding the new transitions csc^+ and csc^-) has been applied in order to resolve the encoding conflict while preserving the consistency of the STG.

6 Conclusions

In this paper, algorithms for checking correctness of several kinds of transition insertions and for performing them directly in the unfolding prefix are presented. The main advantage of the proposed approach is that it avoids re-unfolding. Moreover, it yields a prefix similar to the original one, which is advantageous for visualisation and allows one to transfer some information (e.g., the yet unresolved encoding conflicts) from the original prefix to the modified one.

The algorithms described in this paper have been implemented in our tool MPSat, and successfully applied to resolution of encoding conflicts in STGs [10]. Though some of these algorithms are conservative, in practice good transformations are rarely rejected. In fact, the experimental results conducted in [10] showed that when MPSAT was aiming at optimising the area of the circuit, the resulting circuits were in average 8.8% smaller than those produced by PETRIFY. This is an indication that in this application domain there are usually many available transformations, and so rejecting a small number of them is not detrimental.

In future work, we intend to extend the method to other transformations, in particular concurrency reduction [4].

Acknowledgements. The author would like to thank Walter Vogler for helpful comments. This research was supported by the Royal Academy of Engineering/EPSRC post-doctoral research fellowship EP/C53400X/1 (DAVAC).

References

1. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35-8, 677–691 (1986)
2. Chu, T.-A.: Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (1987)
3. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: PETRIFY: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. IEICE Transactions on Information and Systems E80-D(3), 315–325 (1997)
4. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Automatic Handshake Expansion and Reshuffling Using Concurrency Reduction. In: Proc. HWPN'98, pp. 86–110 (1998)
5. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Logic Synthesis of Asynchronous Controllers and Interfaces. Springer, Heidelberg (2002)
6. Esparza, J., Römer, S., Vogler, W.: An Improvement of McMillan's Unfolding Algorithm. Formal Methods in System Design 20(3), 285–310 (2002)
7. Heljanko, K.: Deadlock and Reachability Checking with Finite Complete Prefixes. Technical Report A56, Laboratory for Theoretical Computer Science, Helsinki University of Technology (1999)
8. Khomenko, V.: Model Checking Based on Prefixes of Petri Net Unfoldings. PhD thesis, School of Computing Science, Newcastle University (2003)
9. Khomenko, V.: Behaviour-Preserving Transition Insertions in Unfolding Prefixes. Technical Report CS-TR-952, School of Computing Science, Newcastle University (2006) URL: `http://homepages.cs.ncl.ac.uk/victor.khomenko/papers/papers.html`
10. Khomenko, V.: Efficient Automatic Resolution of Encoding Conflicts Using STG Unfoldings. Technical Report CS-TR-995, School of Computing Science, Newcastle University (2007) URL: `http://homepages.cs.ncl.ac.uk/victor.khomenko/papers/papers.html`
11. Khomenko, V., Koutny, M., Vogler, V.: Canonical Prefixes of Petri Net Unfoldings. Acta Informatica 40(2), 95–118 (2003)
12. Khomenko, V., Koutny, M., Yakovlev, A.: Detecting State Coding Conflicts in STG Unfoldings Using SAT. Fundamenta Informaticae 62(2), 1–21 (2004)

13. Khomenko, V., Koutny, M., Yakovlev, A.: Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT. Fundamenta Informaticae 70(1–2), 49–73 (2006)
14. Madalinski, A., Bystrov, A., Khomenko, V., Yakovlev, A.: Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. IEE Proceedings: Computers and Digital Techniques 150(5), 285–293 (2003)
15. Murata, T.: Petri Nets: Properties, Analysis and Applications. In: Proc. of the IEEE, 7(4):541–580 (1989)
16. Silva, M., Teruel, E., Colom, J.M.: Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition Net Systems. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 309–373. Springer, Heidelberg (1998)
17. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
18. Yakovlev, A., Lavagno, L., Sangiovanni-Vincentelli, A.: A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis. Formal Methods in System Design 9(3), 139–188 (1996)

Combining Decomposition and Unfolding for STG Synthesis

Victor Khomenko[1] and Mark Schaefer[2]

[1] School of Computing Science, Newcastle University, UK
victor.khomenko@ncl.ac.uk
[2] Institute of Computer Science, University of Augsburg, Germany
mark.schaefer@informatik.uni-augsburg.de

Abstract. For synthesising efficient asynchronous circuits one has to deal with the state space explosion problem. In this paper, we present a combined approach to alleviate it, based on using Petri net unfoldings and decomposition.

The experimental results show significant improvement in terms of runtime compared with other existing methods.

Keywords: Asynchronous circuit, STG, Petri net, decomposition, unfolding, state space explosion.

1 Introduction

Asynchronous circuits are a promising type of digital circuits. They have lower power consumption and electro-magnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations. The International Technology Roadmap for Semiconductors report on Design [ITR05] predicts that 22% of the designs will be driven by 'handshake clocking' (i.e., asynchronous) in 2013, and this percentage will raise up to 40% in 2020.

Signal Transition Graphs, or STGs [Chu87, CKK$^+$02], are widely used for specifying the behaviour of asynchronous control circuits. They are interpreted Petri nets in which transitions are labelled with the rising and falling edges of circuit signals. An STG specifies which outputs should be performed at a given state and, at the same time, it describes assumptions about the environment, which can send an input only if it is allowed by the STG. We use the *speed-independent* model with the following properties:

– Input and outputs edges can occur in an arbitrary order.
– Wires are considered to have no delay, i.e., a signal edge is received simultaneously by all listeners.
– The circuit must work properly according to its formal description under arbitrary delays of each gate.

Synthesis based on STGs involves: (a) checking sufficient conditions for the implementability of the STG by a logic circuit; (b) modifying, if necessary, the initial STG to make it implementable; and (c) finding appropriate Boolean next-state functions for non-input signals.

A commonly used tool, PETRIFY [CKK$^+$97], performs all these steps automatically, after first constructing the reachability graph of the initial STG specification. To gain efficiency, it uses symbolic (BDD-based [Bry86]) techniques to represent the STG's reachable state space. While this state-space based approach is relatively simple and well-studied, the issue of computational complexity for highly concurrent STGs is quite serious due to the *state space explosion* problem [Val98]; that is, even a relatively small STG can (and often does) yield a very large state space. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive (e.g., PETRIFY often fails to synthesise circuits with more that 25–30 signals), especially if the STG models are not constructed manually by a designer but rather generated automatically from high-level hardware descriptions, such as BALSA [EB02] or TANGRAM [Ber93].

In order to alleviate this problem, Petri net analysis techniques based on causal partial order semantics, in the form of Petri net unfoldings, were applied to circuit synthesis. Since in practice STGs usually exhibit a lot of concurrency, but have rather few choice points, their complete unfolding prefixes are often exponentially smaller than the corresponding state graphs; in fact, in many of the experiments conducted in [Kho03, KKY04] they are just slightly bigger than the original STGs themselves. Therefore, unfolding prefixes are well-suited for both visualisation of an STG's behaviour and alleviating the state space explosion problem. The papers [KKY04, KKY06, MBKY03] present a complete design flow for complex-gate logic synthesis based on Petri net unfoldings, which avoids generating the state graph at all stages, and hence has significant advantage both in memory consumption and in runtime, without affecting the quality of the solutions. Moreover, unfoldings are much more visual than state graphs (the latter are hard to understand due to their large sizes and the tendency to obscure causal relationships and concurrency between the events), which enhances the interaction with the user.

The unfolding-based approach can often synthesise specifications which are by orders of magnitude larger than those which can be synthesised by the state-space based techniques. However, this is still not enough for practical circuits. Hence, we combine the unfolding approach with decomposition. Intuitively, a large STG can be decomposed into several smaller ones, whose joint behaviour is the same as that of the original STG. Then these smaller components can be synthesised, one by one, using the unfolding-based approach. STG decomposition was first presented in [Chu87] for live and safe free-choice nets with injective labelling, and then generalised to STGs with arbitrary structure in [VW02, VK05].

This combined framework can cope with quite large specifications. It has been implemented using a number of tools:

PUNF — a tool for building unfolding prefixes of Petri nets [Kho03].

MPSAT — a tool for verification and synthesis of asynchronous circuits; it uses unfolding prefixes built by PUNF, see [KKY04, KKY06].

DesiJ — a tool for decomposing an STG into smaller components. It implements also the techniques of combining decomposition and unfolding presented in this paper and uses Punf and Mpsat for synthesis of final components and for verification of some properties during decomposition, see [VW02, VK05, SVWK06].

2 Basic Definitions

In this section, we present basic definitions concerning Petri nets and STGs, and recall notions related to unfolding prefixes (see also [ERV02, Kho03, Mur89]).

2.1 Petri Nets

A *net* is a triple $N \overset{\text{df}}{=} (P, T, W)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $W : (P \times T) \cup (T \times P) \to \mathbb{N} = \{0, 1, 2, \dots\}$ is a *weight function.* A *marking* M of N is a multiset of places, i.e., $M : P \to \mathbb{N}$. We adopt the standard rules about drawing nets, viz. places are represented as circles, transitions as boxes, the weight function by arcs, and markings are shown by placing tokens within circles. In addition, the following short-hand notation is used: a transition can be connected directly to another transition if the place 'in the middle of the arc' has exactly one incoming and one outgoing arc (see, e.g., Figs. 1(a)). As usual, ${}^\bullet z \overset{\text{df}}{=} \{y \mid W(y, z) > 0\}$ and $z^\bullet \overset{\text{df}}{=} \{y \mid W(z, y) > 0\}$ denote the *pre-* and *postset* of $z \in P \cup T$, and we define ${}^\bullet Z \overset{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \overset{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. We will assume that ${}^\bullet t \neq \emptyset$, for every $t \in T$. N is *finite* if $P \cup T$ is finite, and *infinite* otherwise. A *net system* or *Petri net* is a tuple $\Sigma \overset{\text{df}}{=} (P, T, W, M_0)$ where (P, T, W) is a finite net and M_0 is an *initial* marking.

A transition $t \in T$ is *enabled* at a marking M, denoted $M[t\rangle$, if, for every $p \in {}^\bullet t$, $M(p) \geq W(p, t)$. Such a transition can be *fired*, leading to the marking M' with $M'(p) \overset{\text{df}}{=} M(p) - W(p, t) + W(t, p)$. We denote this by $M[t\rangle M'$. A finite or infinite sequence $\sigma = t_1 t_2 t_3 \dots$ of transitions is a *firing sequence* of a marking M, denoted $M[\sigma\rangle$, if $M[t_1\rangle M'$ and $\sigma' = t_2 t_3 \dots$ is a firing sequence of M' (an empty sequence of transitions is a firing sequence of any marking). Moreover, σ is a firing sequence of Σ if $M_0[\sigma\rangle$. If σ is finite, $M[\sigma\rangle M'$ denotes that σ is a firing sequence of M reaching the marking M'. A marking M' is *reachable from* M if $M[\sigma\rangle M'$ for some firing sequence σ. M is called *reachable* if it is reachable from M_0; $[M_0\rangle$ denotes the set of all *reachable markings* of Σ. Two distinct transitions t_1 and t_2 are in *(dynamic) conflict* if there is a reachable marking M, such that $M[t_1\rangle$, $M[t_2\rangle$ but for some place p, $W(p, t_1) + W(p, t_2) > M(p)$. A dynamic conflict implies a *structural conflict*, i.e. ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$.

A transition is *dead* if no reachable marking enables it. A transition is *live* if any reachable marking M enables a firing sequence containing it. (Note that being live is a stronger property than being non-dead.) A net system is called *live* if every of its transition is live; it is called *reversible* if the initial marking is reachable from every reachable marking.

A net system Σ is *k-bounded* if, for every reachable marking M and every place $p \in P$, $M(p) \leq k$, *safe* if it is 1-bounded, and *bounded* if it is k-bounded for some $k \in \mathbb{N}$. The set of reachable markings of Σ is finite iff Σ is bounded.

2.2 Signal Transition Graphs

A *Signal Transition Graph (STG)* is a triple $\Gamma \overset{\text{df}}{=} (\Sigma, Z, \ell)$ such that Σ is a net system, Z is a finite set of signals, generating the finite alphabet $Z^{\pm} \overset{\text{df}}{=} Z \times \{+, -\}$ of *signal transition labels*, and $\ell : T \to Z^{\pm} \cup \{\lambda\}$ is a labelling function. The signal transition labels are of the form z^+ or z^-, and denote a transition of a signal $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. We will use the notation $z^{\pm}$ to denote a transition of signal z if we are not particularly interested in its direction. Γ inherits the operational semantics of its underlying net system Σ, including the notions of transition enabling and firing sequences, reachable markings and firing sequences.

We lift the notion of enabledness and firing to transition labels: $M[\ell(t)\rangle\rangle M'$ if $M[t\rangle M'$. This is extended to sequences as usual – deleting λ-labels automatically since λ is the empty word. A sequence ω of elements of $Z^{\pm}$ is called a *trace of a marking* M of Γ if $M[\omega\rangle\rangle$, and a *trace* of Γ if it is a trace of M_0. The *language of* Γ, denoted by $L(\Gamma)$, is the set of all traces of Γ. Γ has a *(dynamic) auto-conflict* if two transitions t_1 and t_2 with $\ell(t_1) = \ell(t_2) \neq \lambda$ are in dynamic conflict.

An STG may initially contain transitions labelled with λ called *dummy transitions*. They are a design simplification and describe no physical reality. Moreover, during the decomposition, certain transitions are labelled with λ at intermediate stages; this relabelling of a transition is called *lambdarising* a transition, and *delambdarising* means to change the label back to the initial value. The set of transitions labelled with a certain signal is frequently identified with the signal itself, e.g., lambdarising signal z means to change the label of all transitions labelled with $z^{\pm}$ to λ.

We associate with the initial marking of Γ a binary vector $v^0 \overset{\text{df}}{=} (v^0_1, \ldots, v^0_{|Z|}) \in \{0,1\}^{|Z|}$, where each v^0_i corresponds to the signal $z_i \in Z$; this vector contains the initial value of each signal. Moreover, with any finite firing sequence σ of Γ we associate an integer *signal change vector* $v^{\sigma} \overset{\text{df}}{=} (v^{\sigma}_1, v^{\sigma}_2, \ldots, v^{\sigma}_{|Z|}) \in \mathbb{Z}^{|Z|}$, so that each v^{σ}_i is the difference between the number of the occurrences of z_i^+–labelled and z_i^-–labelled transitions in σ.

Γ is *consistent*[1] if, for every reachable marking M, all firing sequences σ from M_0 to M have the same *encoding vector* $\mathit{Code}(M)$ equal to $v^0 + v^{\sigma}$, and this vector is binary, i.e., $\mathit{Code}(M) \in \{0,1\}^{|Z|}$. Such a property guarantees that, for every signal $z \in Z$, the STG satisfies the following two conditions: (i) the first occurrence of z in the labelling of any firing sequence of Γ starting from M_0 has the same sign (either rising of falling); and (ii) the transitions corresponding to the rising and falling edges of z alternate in any firing sequence of Γ. In this paper it is assumed that all the STGs considered are consistent. (The consistency

[1] This is a somewhat simplified notion of consistency; see [Sem97] for a more elaborated one, dealing also with certain pathological cases, which are not interesting in practice.

of an STG can easily be checked during the process of building its finite and complete prefix [Sem97]; moreover, all the transformations used in this paper preserve consistency.) We will denote by $Code_z(M)$ the component of $Code(M)$ corresponding to a signal $z \in Z$.

The *state graph* of Γ is a tuple $SG_\Gamma \overset{\mathrm{df}}{=} (S, A, M_0, Code)$ such that: $S \overset{\mathrm{df}}{=} [M_0\rangle$ is the set of *states*; $A \overset{\mathrm{df}}{=} \{M \xrightarrow{\ell(t)} M' \mid M \in [M_0\rangle \wedge M[t\rangle M'\}$ is the set of *state transitions*; M_0 is the *initial state*; and $Code : S \to \{0,1\}^{|Z|}$ is the *state assignment* function, as defined above for markings.

The signals in Z are partitioned into input signals, Z_I, and output signals, Z_O (the latter may also include internal signals). Input signals are assumed to be generated by the environment, while output signals are produced by the logic gates of the circuit. For each signal $z \in Z_O$ we define

$$Out_z(M) \overset{\mathrm{df}}{=} \begin{cases} 1 & \text{if } M[z^\pm\rangle\rangle; \\ 0 & \text{otherwise.} \end{cases}$$

Logic synthesis derives for each output signal $z \in Z_O$ a Boolean *next-state function* Nxt_z defined for every reachable state M of Γ as follows:

$$Nxt_z(M) \overset{\mathrm{df}}{=} Code_z(M) \oplus Out_z(M) \; ,$$

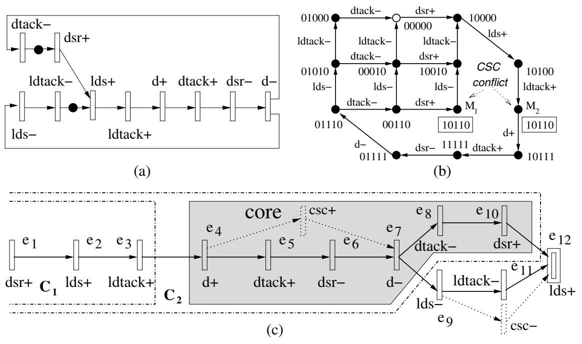
where $\oplus$ is the ‘exclusive or’ operation.

The value of this function must be determined without ambiguity by the encoding of each reachable state, i.e., $Nxt_z(M)$ should be a function of $Code(M)$ rather than of M, i.e., $Nxt_z(M){=}F_z(Code(M))$ for some function $F_z : \{0,1\}^{|Z|} \to \{0,1\}$ (F_z will eventually be implemented as a logic gate). To capture this, let M' and M'' be two distinct states of SG_Γ. M' and M'' are in *Complete State Coding (CSC) conflict* if $Code(M') = Code(M'')$ and $Out_z(M') \neq Out_z(M'')$ for some output signal $z \in Z_O$. Intuitively, a CSC conflict arises when semantically different reachable states of an STG have the same binary encoding. Γ satisfies the *CSC property* if no two states of SG_Γ are in *CSC* conflict. (Intuitively, this means that each output signal is implementable as a logic gate).

An example of an STG for a data read operation in a simple VME bus controller (a standard STG benchmark, see, e.g., [CKK⁺02]) is shown in Figure 1(a). Part (b) of this figure shows a *CSC* conflict between two different states, M_1 and M_2, that have the same encoding, 10110, but $Nxt_d(M_1) = 0 \neq Nxt_d(M_2) = 1$ and $Nxt_{lds}(M_1) = 0 \neq Nxt_{lds}(M_2) = 1$. This means that the values of $F_d(1,0,1,1,0)$ and $F_{lds}(1,0,1,1,0)$ are ill-defined (they should be 0 according to M_1 and 1 according to M_2), and thus these signals are not implementable as logic gates.

2.3 Unfolding Prefixes

A *finite and complete unfolding prefix* of an STG Γ is a finite acyclic net which implicitly represents all the reachable states of Γ together with transitions



$$
\begin{aligned}
[lds] &= \overline{csc}\cdot(\overline{ldtack}\cdot dsr + lds) + d\\
[dtack] &= d\\
[d] &= lds\cdot\overline{csc}\cdot ldtack + d\cdot dsr\\
[csc] &= d + lds\cdot csc
\end{aligned}
$$

(d)

inputs: *dsr*, *ldtack*; **outputs:** *lds*, *d*, *dtack*; **internal:** *csc*

Fig. 1. VME bus controller: the STG for the read cycle (a), its state graph showing a CSC conflict (b), its unfolding prefix with the corresponding conflict core, and a way to resolve it by adding a new signal *csc* (c), and a complex-gate implementation (d). The signal order in binary encodings is: *dsr, dtack, lds, ldtack, d.*

enabled at those states. Intuitively, it can be obtained through *unfolding* Γ, by successive firings of transitions, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated.

The unfolding is infinite whenever Γ has an infinite firing sequence; however, if Γ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. Fig. 1(c) shows a finite and complete unfolding prefix (with the only cut-off event depicted as a double box) of the STG shown in Fig. 1(a).

Efficient algorithms exist for building such prefixes [ERV02, Kho03], which ensure that the number of non-cut-off events in a complete prefix can never exceed the number of reachable states of Γ. However, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself.

Since practical STGs usually exhibit a lot of concurrency, but have rather few choice points, their unfolding prefixes are often exponentially smaller than the corresponding state graphs; in fact, in many of the experiments conducted in [Kho03, KKY04] they were just slightly bigger than the original STGs themselves. Therefore, unfolding prefixes are well-suited for both visualisation of an STG's behaviour and alleviating the state space explosion problem.

3 Unfolding-Based Synthesis

Due to its structural properties (such as acyclicity), the reachable states of an STG can be represented using *configurations* of its unfolding. A configuration C is a downward-closed set of events (being downward-closed means that if $e \in C$ and f is a causal predecessor of e then $f \in C$) without *choices* (i.e., for all distinct events $e, f \in C$, there is no condition c in the unfolding such that the arcs (c, e) and (c, f) are in the unfolding). Intuitively, a configuration is a partially ordered firing sequence, i.e., a firing sequence where the order of firing of some of its events (viz. concurrent ones) is not important.

A CSC conflict can be represented in the unfolding prefix as an unordered *conflict pair* of configurations $\langle C_1, C_2 \rangle$ whose final states are in CSC conflict, as shown in Fig. 1(c). It was shown in [KKY04] that the problem of checking if there is such a conflict pair is reducible to SAT, and an efficient technique for finding all CSC conflict pairs was proposed.

Let $\langle C_1, C_2 \rangle$ be a conflict pair of configurations. The corresponding *complementary set* $\mathcal{CS}$ is defined as the symmetric set difference of C_1 and C_2. $\mathcal{CS}$ is a *core* if it cannot be represented as the union of several disjoint complementary sets. For example, the core corresponding to the conflict pair shown in Fig. 1(c) is $\{e_4, \ldots, e_8, e_{10}\}$ (note that if $C_1 \subset C_2$ then the corresponding complementary set is simply $C_2 \setminus C_1$).

One can show that every complementary set $\mathcal{CS}$ can be partitioned into $C_1 \setminus C_2$ and $C_2 \setminus C_1$, where $\langle C_1, C_2 \rangle$ is a conflict pair corresponding to $\mathcal{CS}$. Moreover, if $C_1 \subset C_2$ then one of these parts is empty, while the other is $\mathcal{CS}$ itself. An important property of complementary sets is that for each signal $z \in Z$, the differences between the numbers of z^+– and z^-–labelled events are the same in these two parts (and are 0 if $C_1 \subset C_2$). This suggests that a complementary set can be eliminated (resolving thus the corresponding encoding conflicts), e.g., by introduction of a new *internal* signal, csc^+, and insertion of its transition into this set, as these would violate the stated property. (Note that the circuit has to implement this new signal, and so for the purpose of logic synthesis it is regarded as an output, though it is ignored by the environment.) To preserve the consistency of the STG, the transition's counterpart, csc^-, must also be inserted *outside the core,* in such a way that it is neither concurrent to nor in structural conflict with csc^+. Another restriction is that an inserted signal transitions must not trigger an input signal transition (the reason is that this would impose constraints on the environment which were not present in the original STG, making it 'wait' for the newly inserted signal). Intuitively, insertion of signals introduces additional memory into the circuit, helping to trace the current state.

The core in Fig. 1(c) can be eliminated by inserting a new signal, csc^+, somewhere in the core, e.g., concurrently to e_5 and e_6 between e_4 and e_7, and by inserting its complement outside the core, e.g., concurrently to e_{11} between e_9 and e_{12}. (Note that the concurrent insertion of these two transitions avoids an increase in the latency of the circuit, where each transition is assumed to contribute a unit delay.) After transferring this signal into the STG, it satisfies the CSC property.

It is often the case that cores overlap. In order to minimise the number of performed transformations, and thus the area and latency of the circuit, it is advantageous to perform such a transformation that as many cores as possible are eliminated by it. That is, a transformation should be performed in the *intersection of several cores* whenever possible.

This idea can be implemented by means of a *height map* showing the quantitative distribution of the cores. Each event in the prefix is assigned an *altitude*, i.e., the number of cores it belongs to. (The analogy with a topographical map showing the altitudes may be helpful here.) 'Peaks' with the highest altitude are good candidates for insertion, since they correspond to the intersection of maximum number of cores. This unfolding-based method for the resolution of encoding conflicts was presented in [MBKY03].

Once the CSC conflicts are resolved, one can derive equations for logic gates of the circuit, as illustrated in Fig. 1(d). An unfolding-based approach to this problem has been presented in [KKY06]. The main idea of this approach was to generate the truth table for each such equation as a projection of a set of reachable encodings to some chosen support, which can be accomplished with the help of the incremental SAT technique, and then applying the usual Boolean minimisation to this table.

The results in [KKY04, MBKY03, KKY06] form a complete design flow for complex-gate synthesis of asynchronous circuits based on STG unfoldings rather than state graphs, and the experimental results conducted there show that it has significant advantage both in memory consumption and in runtime, without affecting the quality of the solutions.

4 STG Decomposition

In this section, the STG decomposition algorithm of [VW02, VK05] is outlined, in order to understand the new contributions properly.

Synthesis with STG decomposition works roughly as follows. Given a consistent STG $\mathit{\Gamma}$, an *initial partition* $(In_i, Out_i)_{i \in I}$ of its signals is chosen, satisfying the following properties.

- $(Out_i)_{i \in I}$ is a partition of the output signals of the original STG (the sets In_i may overlap with In_j and Out_j if $i \neq j$).
- If two output signals x_1, x_2 are in structural conflict in $\mathit{\Gamma}$, then they have to be in the same Out_i.
- If there are $t, t' \in T$ with $t' \in (t^\bullet)^\bullet$ (t is called *syntactical trigger of* t'), then $\ell(t') \in Out_i$ implies $\ell(t) \in In_i \cup Out_i$.

Then the algorithm decomposes $\mathit{\Gamma}$ into component STGs, one for each element in this partition, together implementing $\mathit{\Gamma}$. Each component is obtained from the original STG by lambdarising the signals which are not in the corresponding element of the partition, and then contracting the corresponding transitions (some other net reductions are also applied — see below). Then, from each component a circuit is synthesised, and these circuits together implement the original specification.

Of course, the decomposition must preserve the behaviour of the specification in some sense. In [VW02, VK05, SV05], the correctness was defined as a variation of bisimulation, tailored to the specific needs of asynchronous circuits, called *STG-bisimulation*.

Typically, the computational effort (in terms of memory consumption and runtime) needed to synthesise a circuit from an STG Γ is exponential in the size of Γ. Hence, if the components produced by the decomposition algorithm are smaller than Γ, the decomposition can be seen as successful. (Note that in the worst case the components can be as large as the original STG, but this rarely happens in practice).

We now describe the operations which the algorithm applies to an initial component until no more λ-labelled transitions remain.

Contraction of a λ-labelled transition. Transition contraction can be applied to a λ-labelled transition t if ${}^\bullet t \cap t^\bullet = \emptyset$, and for each place p, $W(t,p), W(p,t) \leq 1$; it is illustrated in Figure 2. Intuitively, t is removed from the net, together with its surrounding places ${}^\bullet t \cup t^\bullet$, and the new places, corresponding to the elements of ${}^\bullet t \times t^\bullet$, are added to the net. Each new place $(p,q) \in {}^\bullet t \times t^\bullet$ inherits the connectivity of both p and q (except that t is no longer in the net), and its initial marking is the total number of tokens which were initially present in p and q. (The formal definition of transition contraction can be found in [VW02, VK05, KS06]).

The contraction is called *secure* if either $({}^\bullet t)^\bullet \subseteq \{t\}$ (*type-1 secure*) or ${}^\bullet(t^\bullet) = \{t\}$ and $M_0(p) = 0$ for some $p \in t^\bullet$ (*type-2 secure*). It is shown in [VW02, VK05] that secure contractions of λ-labelled transitions preserve the language of the STG.

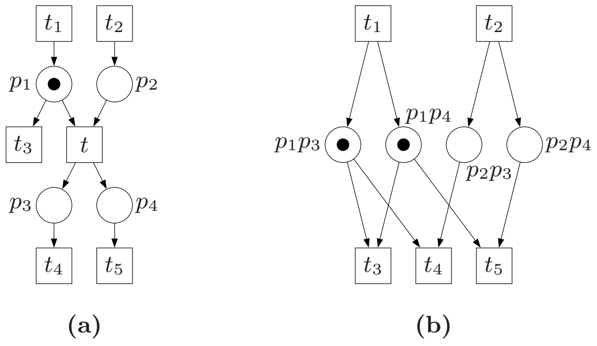


Fig. 2. Transition contraction: initial net **(a)**, and the one after contraction of t **(b)**

Deletion of an implicit place. It is often the case that after a transition contraction *implicit* places (i.e., ones which can be removed without changing the firing sequences of the net) are produced. Such places may prevent further transition contractions, and should be deleted before the algorithm proceeds.

Deletion of a redundant transition. There are two kinds of *redundant transitions*. First, if there are two transitions with the same label which are connected to every place in the same way, one of them can be deleted without changing the traces of the STG. Second, a λ-labelled transition t with ${}^\bullet t = t^\bullet$ can also be deleted, since its firing does not change the marking and is not visible on the level of traces; observe, that this is valid for any marking of the adjacent places.

Backtracking. As it was already mentioned, not every λ-labelled transition can be contracted by the decomposition algorithm. There are three possible reasons for this:

– The contraction is not defined (e.g., because ${}^\bullet t \cap t^\bullet \neq \emptyset$).
– The contraction is not *secure* (then the language of the STG might change).
– The contraction introduces a new auto-conflict (i.e., a new potential source of non-determinism which was not present in the specification is introduced; this is interpreted that the component has not enough information (viz. input signals) to properly produce its outputs).

If none of the described reduction operations are applicable, but the component still has some λ-labelled transitions, *backtracking* is applied, i.e., one of these λ-labelled transitions is chosen and the corresponding signal is *delambdarised*, i.e., this input is added to the initial partition and the new corresponding initial component is derived and reduced from the beginning. This cycle of reduction and backtracking is repeated until all λ-labelled transitions of the initial component can be contracted. This means that backtracking is only needed to detect these additional input signals; if they are known in advance, one can perform decomposition completely without backtracking. (In the worst case, all the lambdarised signals are delambdarised).

The described decomposition algorithm is non-deterministic, i.e., it can apply the net reductions in any order; the result has been proven to be always correct. In [SVWK06], different ways to determinise it are described. One of them was *tree decomposition*, which greatly improves the overall efficiency of decomposition process by re-using intermediate results. Since it is the base for CSC-aware decomposition introduced below, we describe it briefly.

4.1 Tree Decomposition

In our experiments, it turned out that in most cases some initial components have many lambdarised signals in common. Therefore, the decomposition algorithm can save time by building an intermediate STG C', from which these components can be derived: instead of reducing both initial components independently, it is sufficient to generate C' only once and to proceed separately with each component afterwards, thus saving a lot of work.

Tree decomposition tries to generate a plan which minimises the total amount of work using the described idea. We introduce it by means of an example in Figure 3. Let Γ be an STG with the signal set $\{1, 2, 3, 4, 5\}$. Furthermore, let there be three components C_1, C_2, C_3, and let $\{1, 2, 3\}$, $\{2, 3, 4\}$, $\{3, 4, 5\}$ be the signals which

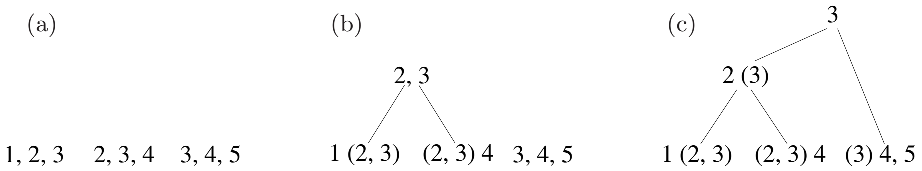


Fig. 3. Building of a simple decomposition tree for three components and five signals. Leafs from the left: components C_1, C_2, C_3. (a) the initial situation; (b) two components merged; (c) the final decomposition tree.

are lambdarised initially in these components. We build a tree guiding the decomposition process, such that its leafs correspond to the final components, and every node u is labelled with the set of signals $s(u)$ to be contracted.

In (a) the initial situation is depicted. There are three independent leaves labelled with the signals which should be contracted to get the corresponding final component. A possible intermediate STG C' for C_1 and C_2 would be the STG in which signals 2 and 3 have been contracted. In (b), C' is introduced as an common intermediate result for C_1 and C_2; the signals 2 and 3 no longer have to be contracted in C_1 and C_2 (they appear in brackets) and the leaves are labelled with $\{1\}$ and $\{4\}$, respectively. In (c), a common intermediate result for C' and C_3 with the label $\{3\}$ is added, yielding the final decomposition tree.

From this use of a decomposition tree, it is clear that in an optimal decomposition tree the sum of all $|s(u)|$ should be minimal. Decomposition trees are very similar to *preset trees* in [KK01]; there it is shown that computing an optimal preset tree is NP-complete, and a heuristic algorithm is described which performs reasonably well. We use this algorithm for the automatic calculation of decomposition trees.

The decomposition algorithm guided by such a decomposition tree traverses it in the depth-first order. It enters the root node with the initial STG Γ containing no lambdarised signals. Upon entering a node u with an STG Γ_u, the algorithm lambdarises and contracts the signals $s(u)$ in Γ_u (and performs other possible reductions) and enters each child node with its own copy of the resulting STG.[2] If u is a leaf, the resulting STG is a final component.[3]

4.2 CSC-Aware Decomposition

On the basis of tree decomposition, we now introduce *CSC-aware* decomposition. Our aim is to reduce the number of CSC conflicts in the components generated

[2] As an important technical improvement, the intermediate result of a component is not copied for each child. Instead, throughout the decomposition, a single STG is held in memory, and an *undo stack* is used to restore the 'parent' STG whenever the algorithm returns to the parent node. This is much faster and uses far less memory than keeping multiple (and potentially large) STGs.

[3] There are some twists in this setting considering backtracking, which is handled a bit different in contrast to 'ordinary' decomposition; in particular, the decomposition tree can be modified during the decomposition process, cf. [SVWK06].

by the decomposition algorithm. Ideally, if the original specification is free from CSC conflicts then this should be the case also for the components.

During its execution the algorithm has to determine if an STG has CSC conflicts. This is checked externally with Punf and Mpsat [Kho03, KKY04]. It works essentially as tree decomposition, with the following differences, cf. Figure 4. When a leaf is reached, we check whether the corresponding final component has CSC conflicts. If no, the component is saved as the final result. Otherwise, for each detected CSC core a constituting pair of firing sequences leading to the conflicting states is stored in the parent of the leaf.

When the algorithm returns to this parent node, it checks whether this CSC conflict is still present in the local intermediate STG. However, using Mpsat may be expensive at this point, as the corresponding STG is larger than the final component. Instead, we map the stored firing sequences from the final component to this STG using the *inverse projections* introduced below, and check if they still lead to states which are in a CSC conflict. For every conflict which is not destroyed, this results in a new pair of firing sequences which is propagated upwards in the tree, and so on. On the other hand, if the conflict disappears, these inverse projections are analysed as described below, and signals which helped to resolve the conflict are determined and delambdarised in the corresponding child node, and the algorithm tries to process it again. If no CSC conflicts remain in the final component (due to the delambdarised signals), it is saved as the final result.

When all pairs of firing sequences corresponding to CSC conflicts are considered, the algorithm proceeds with the next child of the current node. If there are no more children left, it goes up to the parent of the current node and deals with the corresponding firing sequences. Eventually, the algorithm reaches the root node for the last time and terminates.

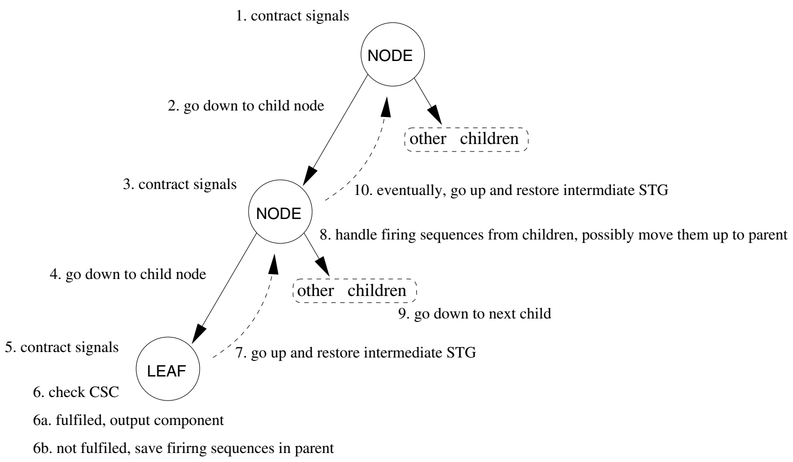

1. contract signals
NODE
2. go down to child node
other children
3. contract signals
10. eventually, go up and restore intermdiate STG
NODE
8. handle firing sequences from children, possibly move them up to parent
4. go down to child node
other children
9. go down to next child
5. contract signals
7. go up and restore intermediate STG
LEAF
6. check CSC
6a. fulfiled, output component
6b. not fulfiled, save firirng sequences in parent

Fig. 4. Outline for CSC-aware decomposition. Step 8 is repeated every time a node is entered from a child, step 9 includes contraction and detection of CSC conflicts.

This algorithm is complete, i.e., it guarantees for a specification with CSC that each component has CSC, too. This is due to the fact that a pair of firing sequences corresponding to a CSC conflict in a component can be moved up to the root node (via a sequence of inverse projections), where CSC is given initially. In practice, one can stop moving up a pair of firing sequences after several iterations and try to resolve CSC conflicts with new internal signals instead. Therefore, the algorithm is still applicable to specifications which have CSC conflicts initially.

The *inverse projection* of a firing sequence is defined as follows. Let Γ and Γ' be two STGs such that Γ' is obtained from Γ by a secure contraction of some transition t. If σ' is a firing sequence of Γ', we call a firing sequence σ of Γ an inverse projection of σ' if σ' is the projection of σ on the transitions of Γ'.

Since the contraction of t was secure, the inverse projection can be calculated easily: it is enough to fire the transitions of σ' in Γ, one by one, while possible. If, at some point, a transition of σ' cannot be fired then t is fired (it is guaranteed to be enabled in such a case). This process is continued until all the transitions of σ' are fired, yielding σ. One can see that a shortest inverse projection is computed by the described procedure.

If Γ' is obtained from Γ by a sequence of secure contractions, its firing sequences can still be inversely projected to Γ by computing a sequence of inverse projections for each individual contraction.

If Γ' has a CSC conflict, there is a corresponding pair of firing sequences (σ'_1, σ'_2) such that the corresponding signal change vectors $v^{\sigma'_1}$ and $v^{\sigma'_2}$ coincide. If the inverse projection (σ_1, σ_2) of this pair is such that $v^{\sigma_1} \neq v^{\sigma_2}$ then the corresponding conflict is likely to be destroyed by delambdarising the signal corresponding to the contracted transition.

5 Combining Decomposition and Unfolding Techniques

In this section we describe how our unfolding and decomposition tools can be used to combine their advantages and to compensate for each other's shortcomings. Punf and Mpsat can perform logic synthesis, but not for very large STGs. On the other hand, DesiJ can handle very large STGs quite efficiently because it performs only local structural operations, but it has to make conservative assumptions frequently to guarantee correctness.

The strategy we adopted is as follows. While the STGs are large, only structural conservative checks are made, as it may be computationally very expensive to perform the exact tests. After some reductions have been performed, it becomes feasible to check exact reachability-like properties using Punf and Mpsat (logic synthesis is still not feasible at this stage). Eventually, when the components are small enough, logic synthesis is performed.

While DesiJ can handle and produce non-safe nets, Punf and Mpsat need safe nets. Therefore, we accept only safe nets as specifications (which is no serious restriction) and perform only *safeness-preserving* contractions during decomposition. They are discussed in the following subsection.

During the decomposition process the decomposition algorithm checks from time to time the following reachability-like properties:

- The decomposition algorithm should backtrack if a new dynamic auto-conflict is produced. The corresponding conservative test is the presence of a new structural auto-conflict.
- It is also helpful to remove implicit places. The corresponding conservative test looks for *redundant places* [Ber87]; they are defined by a system of linear inequalities. Checking this condition with a linear programming solver is also quite expensive, and therefore DesiJ looks only for a subset called *shortcut places* [SVJ05].
- In order to apply Mpsat, the STG must be safe. In general, a transition contraction can transform a safe STG into an non-safe (2-bounded) one. The corresponding conservative structural conditions guaranteeing that a contraction preserves safeness are developed below.

If the STG is not too large, all of the mentioned dynamic properties can also be checked exactly with a reachability analysis. Since we only consider safe nets here, reachability-like properties can be expressed as Boolean expressions over the places of the net. For example, the property $p_1 \wedge p_2 \wedge \neg p_3$ holds iff some reachable marking has a token in p_1 and p_2 and no token in p_3. (Such properties can be checked by Mpsat.) Below we give Boolean expressions and the corresponding conservative tests for the properties listed above.

Safeness-Preserving Contractions

A transition contraction preserves boundedness, but, in general, it can turn a safe net into a non-safe one, as well as introduce duplicate (weighted) arcs. However, since unfolding techniques are not very efficient for non-safe net, we assume that the initial STG is safe, and perform only *safeness-preserving* contractions, i.e., ones which guarantee that if the initial STG was safe then the transformed one is also safe. (Note that the transitions with duplicate (weighted) arcs must be dead in a safe Petri net, and so we can assume that the initial and all the intermediate STGs contain no such arcs).

We now give a sufficient structural condition for a contraction being safeness-preserving. Then we will show how this can be checked with a reachability analysis and also how a single unfolding prefix can be used for checking if each contraction in a sequence of contractions is safeness-preserving. (The proofs of all the results can be found in the technical report [KS06]).

Theorem 1 (Structural safeness-preservation). *A secure contraction of a transition t in a net Γ is safeness-preserving if*

1) $|{}^\bullet t| = 1$ *or*
2) $|t^\bullet| = 1$, ${}^\bullet(t^\bullet) = \{t\}$ *and*
 a) Γ *is live and reversible*
 or
 b) $M_0(p) = 0$ *with* $t^\bullet = \{p\}$

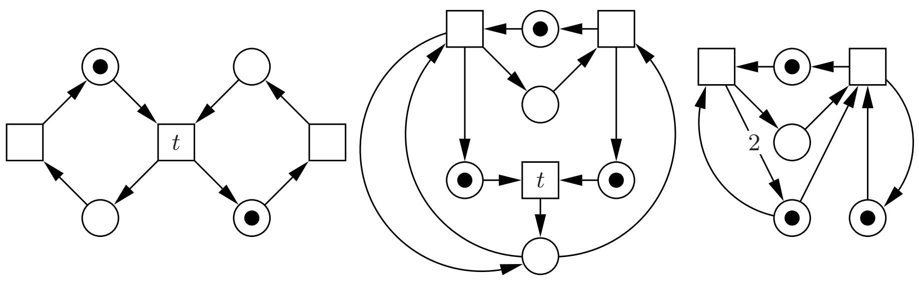


Fig. 5. Examples of non-safeness-preserving contractions

Figure 5 shows two counterexamples: the leftmost net violates the condition that either the pre- or postset of t has to contain a single place; one can see that the contraction of t generates a non-safe net. The net in the middle violates the condition $^\bullet(t^\bullet) = \{t\}$ in the second case in Theorem 1 (i.e., that the place in the postset of t must not have incoming arcs other than from t); the rightmost net is obtained by contracting t in the net in the middle.

In practice, the decomposition algorithm checks the condition 2b) which makes no assumptions about the net which are difficult to verify. This is important since there exist STGs which are neither live nor reversible, e.g., ones which have some initialisation part which is executed only once in the beginning.

If the specification is guaranteed to be live and reversible, it is also possible to use condition 2a); then the following lemma is needed to apply such contractions repeatedly.

Proposition 2. *Secure transition contractions and implicit place deletions preserve liveness and reversibility.*

So far, we only considered structural conditions for a contraction to be safeness-preserving; now we describe the dynamic conditions.

Theorem 3. *Let Γ be a safe STG and $t \in T$ such that the contraction of t is secure. The contraction of t is safeness-preserving iff the following property does not hold:*

$$\left(\bigvee_{p\in{}^\bullet t} p\right) \wedge \left(\bigvee_{p\in t^\bullet} p\right) .$$

To check these reachability properties with MPSAT one has to generate the unfolding prefix with PUNF first, which can take considerable time. It is therefore impractical to generate it for checking the safeness-preservation of a single contraction. Instead, our algorithm uses a single unfolding prefix to check if a sequence of several subsequent contractions is safeness-preserving. (This technique is described in more detail in the technical report [KS06]).

Implicit Places

As it was already mentioned, the deletion of implicit places is important for the success of the decomposition. As a conservative condition, DesiJ looks for shortcut places. On the other hand, unfolding-based reachability analysis makes it possible to check exactly whether a place is implicit: a place p is implicit iff the following property does not hold:

$$\neg p \wedge \left(\bigvee_{t \in p^\bullet} \bigwedge_{q \in {}^\bullet t \setminus \{p\}} q \right) .$$

It is possible to detect all implicit place of a net with a single unfolding. Observe first, that the deletion of an implicit place cannot turn a non-implicit place into an implicit one. Indeed, suppose p_1 is implicit and deleted in Σ, yielding Σ_1, and p_2 is implicit and deleted in Σ_1, yielding Σ_2. Then $FS(\Sigma) = FS(\Sigma_1) = FS(\Sigma_2)$ by definition of implicit places, where $FS(\Sigma)$ denotes the set of all firing sequences of Σ. Suppose now that p_2 is deleted first in Σ, yielding Σ_1', and p_1 is deleted in Σ_1', yielding Σ_2 again. Then $FS(\Sigma) \subseteq FS(\Sigma_1') \subseteq FS(\Sigma_2) = FS(\Sigma)$, since deleting places can only increase the set of firing sequences. Therefore $FS(\Sigma) = FS(\Sigma_1') = FS(\Sigma_2)$, which shows that p_2 is implicit in Σ. It is therefore sufficient to iterate once over all places and to delete every implicit one.

Furthermore, the unfolding of a net in which an implicit place was deleted can be obtained from the original unfolding by deleting all occurrences of this place. For the above reachability analysis we get the same effect automatically, because deleted places will not occur in the corresponding property.

Dynamic Auto-conflicts

A conservative test for the presence of an auto-conflict is the presence of two transitions with the same label (distinct from λ) and overlapping presets. Unfolding-based reachability analysis makes it possible to check exactly for the presence of an auto-conflict as follows.

In a safe STG distinct transitions t_1 and t_2 such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ are in dynamic conflict iff the following property holds:

$$\bigwedge_{p \in {}^\bullet t_1 \cup {}^\bullet t_2} p .$$

Using this exact test can reduce the number of times the decomposition algorithm has to backtrack, which ultimately can result in the improved runtime and smaller final components.

6 Results

We applied the described combined approach to several benchmark examples with and without CSC conflicts, and compared the results with the stand-alone synthesis with Mpsat and Petrify. (The tool for CSC conflict resolution and decomposition described in [CC06, Car03] was not available from the authors.) In the tables, all times are given as (minutes:)seconds. The benchmarks were performed on a Pentium 4 HT with 3 GHz and 2 GB RAM.

We worked with two types of benchmarks. The first group are pipelines which have CSC initially. As expected, the new approach produces components without CSC conflicts, i.e., the signals which are necessary for preventing CSC conflicts are kept in the components (the original approach of [VW02, VK05, SVWK06] would have contracted some of them).

Our combined approach decomposes and synthesises these benchmarks (see Table 1) quite quickly compared with Petrify (aborted after 6 hours). However, Mpsat alone is much faster for these examples and needs less than a second for any of them. This is because these benchmarks are relatively small, with up to 257 nodes and up to 43 signals.

Table 1. Results for the pipeline benchmarks

Benchmark	DesiJ	Petrify
2pp.arb.nch.03.csc	1	1
2pp.arb.nch.06.csc	2	14
2pp.arb.nch.09.csc	4	1:54
2pp.arb.nch.12.csc	10	32:55
2pp-wk.03.csc	1	1
2pp-wk.06.csc	2	9
2pp-wk.09.csc	3	31
2pp-wk.12.csc	18	24:36
3pp.arb.nch.03.csc	1	4
3pp.arb.nch.06.csc	3	2:14
3pp.arb.nch.09.csc	7	84:17
3pp.arb.nch.12.csc	22	$\geq$ 360:00
3pp-wk.03.csc	1	1
3pp-wk.06.csc	3	31
3pp-wk.09.csc	7	34:08
3pp-wk.12.csc	22	$\geq$ 360:00

The second group of benchmarks are newly generated; they are STGs derived from Balsa specifications. These kind of benchmarks was used before by [CC06]. The benchmark SeqParTree(21,10) from there is nearly the same as SeqParTree-05 here; the difference is that we did not hide the internal handshake signals. However, this is also possible for our approach and will most likely lead to further speedups, as discussed in Section 7.

These examples are generated out of two basic Balsa handshake components (see [EB02]): the 2-way *sequencer*, which performs two subsequent handshakes on its two 'child' ports when activated on its 'parent' port, and the 2-way

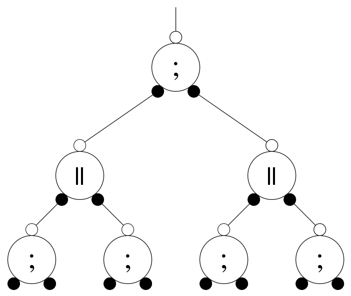


Fig. 6. SEQPARTREE-03. Filled dots denote active handshake ports (they can start a handshake), blank nodes denote passive ones. Each port is implemented by two signals, an input and an output. If two ports are connected the parallel composition merges these four signals into two outputs.

Table 2. Results of the handshake benchmarks

	Size	Signals	Combined		
Benchmark	$\|P\| - \|T\|$	$\|In\| - \|Out\|$	Deco.	Synthesis	Σ
SEQPARTREE-05	382 – 252	33 – 93	2	1	3
SEQPARTREE-06	798 – 508	65 – 189	4	2	6
SEQPARTREE-07	1566 – 1020	129 – 381	10	4	14
SEQPARTREE-08	3230 – 2044	257 – 765	48	10	57
SEQPARTREE-09	6302 – 4092	513 – 1533	4:55	24	5:19
SEQPARTREE-10	12958 – 8188	1025 – 3069	68:09	1:39	69:48

paralleliser, which performs two parallel handshakes on its two 'child' ports when activated on its 'parent' port; either can be described by a simple STG. The benchmark examples SEQPARTREE-N are complete binary trees with alternating levels of sequencers and parallelisers, as illustrated in Figure 6 (N is the height of the tree), which are generated by the parallel composition of the elementary STGs corresponding to the individual sequencers and parallelisers in the tree. We also worked with other benchmarks made of handshake components (e.g., trees of parallelisers only); the results did not differ much, so we considered exemplarily only SEQPARTREE-N.

These benchmarks have CSC conflicts initially, and MPSAT was used in the end to resolve them in each component separately. The experimental results in Table 2 show the real power of our method. The corresponding STGs are very large, and we consider it as a important achievement that the proposed combined approach could synthesise them so quickly. As one can see, an STG with more than 4000 signals is synthesised in less than 70 minutes. PETRIFY and MPSAT alone need more than 12 hours (aborted) for either of these benchmarks.

In contrast to the decomposition method of [CC03, CC06] we allow components with more than output. This was utilised here: the initial partition was chosen such that each component of the decomposition corresponds to one handshake component. Other partitions of the outputs might lead to further speedups.

7 Conclusion

The purely structural decomposition approach of [VW02, VK05, SVWK06] can handle large specifications, but it does not take into account the properties of STGs related to synthesisability, such as the presence of CSC conflicts. In contrast, MPSAT can resolve CSC conflicts and perform logic synthesis, but it is inefficient for large specifications. In this paper, we demonstrated how these two methods can be combined to synthesise large STGs very efficiently.

One of the main technical contributions was to preserve the safeness of the STGs throughout the decomposition, because MPSAT can only deal with safe STGs. This is not just an implementation issue or a compensation for a missing MPSAT feature, but it is also far more efficient than working with non-safe nets, for which unfolding techniques seem to be inefficient. We also showed how dynamic properties like implicitness and auto-conflicts can be checked with unfoldings and how these checks can be combined with cheaper conservative structural conditions.

Future research is required for the calculation of the decomposition tree, the size of which is cubic in the number of signals and exceeds the memory usage for decomposition and synthesis by far. Here, heuristics are needed which explore the tradeoff between the quality of the decomposition tree and the amount of memory needed for its calculation.

Furthermore, we consider the handling of handshake based STGs as very important. Handshake circuits allow to synthesise very large specifications at the expense of a heavy overencoding of the resulting circuit, i.e., they have a lot of unnecessary state-holding elements, which increase the circuit area and latency. Decomposition can help here in the following way: instead of synthesising each handshake component separately, one can combine several such components, e.g., as it was done for SEQPARTREE-N, hide the internal communication signals and synthesise one circuit implementing the combination of the components using the proposed combined approach.

Acknowledgements. We would like to thank Dominic Wist for helping us generating the benchmarks. This research was supported by DFG-projects 'STG-Dekomposition' Vo615/7-1 and Wo814/1-1, and the Royal Academy of Engineering/EPSRC grant EP/C53400X/1 (DAVAC).

References

[Ber87] Berthelot, G.: Transformations and decompositions of nets. In: Brauer, W., Reisig, R., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties. LNCS, vol. 254, pp. 359–376. Springer, Heidelberg (1987)

[Ber93] Berkel, K.v.: Handshake Circuits: an Asynchronous Architecture for VLSI Programming. International Series on Parallel Computation, vol. 5 (1993)

[Bry86] Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35-8, 677–691 (1986)

[Car03] Carmona, J.: Structural Methods for the Synthesis of Well-Formed Concurrent Specifications. PhD thesis, Univ. Politècnica de Catalunya (2003)

[CC03] Carmona, J., Cortadella, J.: ILP models for the synthesis of asynchronous control circuits. In: Proc. of the IEEE/ACM International Conference on Computer Aided Design, pp. 818–825 (2003)

[CC06] Carmona, J., Cortadella, J.: State encoding of large asynchronous controllers. In: DAC 2006, pp. 939–944 (2006)

[Chu87] Chu, T.-A.: Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications. PhD thesis, MIT (1987)

[CKK$^+$97] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Trans. Information and Systems E80-D, 3, 315–325 (1997)

[CKK$^+$02] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Logic Synthesis of Asynchronous Controllers and Interfaces. Springer, Heidelberg (2002)

[EB02] Edwards, D., Bardsley, A.: BALSA: an Asynchronous Hardware Synthesis Language. The Computer Journal 45(1), 12–18 (2002)

[ERV02] Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods in System Design 20(3), 285–310 (2002)

[ITR05] International technology roadmap for semiconductors: Design (2005) URL: `www.itrs.net/Links/2005ITRS/Design2005.pdf`

[Kho03] Khomenko, V.: Model Checking Based on Prefixes of Petri Net Unfoldings. PhD thesis, School of Computing Science, Newcastle University (2003)

[KK01] Khomenko, V., Koutny, M.: Towards an efficient algorithm for unfolding Petri nets. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, Springer, Heidelberg (2001)

[KKY04] Khomenko, V., Koutny, M., Yakovlev, A.: Detecting state coding conflicts in STG unfoldings using SAT. Fundamenta Informaticae 62(2), 1–21 (2004)

[KKY06] Khomenko, V., Koutny, M., Yakovlev, A.: Logic synthesis for asynchronous circuits based on Petri net unfoldings and incremental SAT. Fundamenta Informaticae 70(1–2), 49–73 (2006)

[KS06] Khomenko, V., Schaefer, M.: Combining decomposition and unfolding for STG synthesis. Technical Report 2006-01, University of Augsburg (2006) URL: `http://www.Informatik.Uni-Augsburg.DE/skripts/techreports/`

[MBKY03] Madalinski, A., Bystrov, A., Khomenko, V., Yakovlev, A.: Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. IEE Proceedings: Computers and Digital Techniques 150(5), 285–293 (2003)

[Mur89] Murata, T.: Petri Nets: Properties, Analysis and Applications. Proc. of the IEEE 77(4), 541–580 (1989)

[Sem97] Semenov, A.: Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding. PhD thesis, Newcastle University (1997)

[SV05] Schaefer, M., Vogler, W.: Component refinement and CSC solving for STG decomposition. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 348–363. Springer, Heidelberg (2005)

[SVJ05] Schaefer, M., Vogler, W., Jančar, P.: Determinate STG decomposition of marked graphs. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 365–384. Springer, Heidelberg (2005)

[SVWK06] Schaefer, M., Vogler, W., Wollowski, R., Khomenko, V.: Strategies for optimised STG decomposition. In: Proc. of ACSD (2006)

[Val98] Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)

[VK05] Vogler, W., Kangsah, B.: Improved decomposition of signal transition graphs. In: ACSD 2005, pp. 244–253 (2005)

[VW02] Vogler, W., Wollowski, R.: Decomposition in asynchronous circuit design. In: Cortadella, J., Yakovlev, A.V., Rozenberg, G. (eds.) Concurrency and Hardware Design. LNCS, vol. 2549, pp. 152–190. Springer, Heidelberg (2002)

Object Nets for Mobility

Michael Köhler and Berndt Farwer

University of Hamburg, Department for Informatics
koehler@informatik.uni-hamburg.de
University of Durham, Department of Computer Science
berndt.farwer@durham.ac.uk

Abstract. This paper studies mobile agents that act in a distributed name space. The difference between belonging to a name space (where objects can be accessed directly via pointers) and migrating between name spaces (where objects have to be treated as values, that can be copied into network messages) is taken account of by introducing Petri net based formalism, employing the *nets-within-nets* paradigm.

This formalism, called *mobile object nets*, generalises the well-established theory of elementary object nets, which has seen many applications over the last decade.

While mobile object nets provide a solution to the specific modelling problem mentioned above, they are much more generic and not restricted to this domain.

Keywords: Mobile agents, name spaces, nets-within-nets, object nets, Petri nets.

1 Introduction

Mobility induces new challenges for dynamic systems, which need a new conceptional treatment. In this article we formalise mobile agents acting in a mobility infrastructure like that in Figure 1. In this figure, the infrastructure is composed of the two buildings A and B represented in the system net. Buildings can be seen as a metaphor, e.g., for different hosts on a distributed network. The two buildings are connected via the mobility transfer transitions t_4 and t_6. One mobile agent is present inside building A as a net token.

Inside the building the agent has access to a workflow describing how the agent is allowed to use services, i.e. the building's infrastructure. The agent can decide to use the building's infrastructure by synchronising with the access workflow's transitions. The transition inscriptions (given in the syntax of the tool Renew [KWD+04]) generate the set of synchronisations (t_2, t_{20}), (t_3, t_{21}), (t_5, t_{22}), and (t_7, t_{23}).

The initial marking $\mu_0 = (p_1, p_{20})$ is a nested multiset, which reflects the fact that we model a hierarchical scenario (an agent located *inside* a building). In this scenario the mobile agent's net is copied to the places p_2 and p_3 by execution of transition t_1. The agent's initial marking $M_0 = p_{20}$ disables all events except for the synchronisation (t_2, t_{20}). After the synchronisation the agent's marking

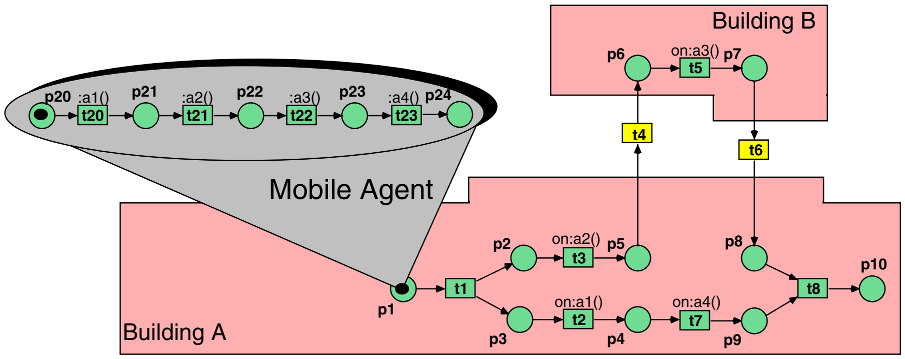


Fig. 1. A mobile agent's environment

becomes $M_1 = p_{21}$ which enables the synchronisation (t_3, t_{21}). The agent can now travel from building A to B via transition t_4. The arrival of the agent at building B enables (t_5, t_{22}). At this point the agent's marking has evolved to $M_2 = p_{23}$ and in building A the place p_4 is marked. Nevertheless, this is not sufficient to enable (t_7, t_{23}), since we have to take into account that the agent's token p_{23} is still at building B. This token is unavailable until the agent moves back to building A via t_6. When the agent is back inside the building A at place p_8 then (t_7, t_{23}) is enabled. After firing t_8 the final marking $\mu = (p_{10}.p_{24})$ is reached.

When modelling this scenario we have to distinguish two kinds of movement: Movement within a building and movement from one building to another. When moving whithin a building, the agent has full access to all services (e.g. service stations, information servers, etc.). On the other hand, when moving to a different building the environment may change dramatically: Services may become unavailable, they may change their name or their kind of access protocols. This leads to the usual problem that within the same environment (e.g. the memory of a personal computer) we can use pointers to access objects (as done for Java objects), which is obviously impossible for a distributed space like a computer network: For example when a Java program transfers an object from machine A to B via remote method invocation (RMI) it does not transfer the object's pointers (which are not valid for B); instead Java rather makes a *deep copy* of the object (called *serialisation*) and transfers this value over the network. The value is used to generate a new object at B which can be accessed by a fresh pointer.

Among the wealth of research on defining mobile systems, in recent years a variety of formalisms have been introduced or adopted to cover mobility: The approaches can be roughly separated into process calculi and Petri net based approaches. The π-calculus [MPW92], the Ambient-calculus [CGG00] and the Seal calculus [VC98] are just three of the more popular calculi. Approaches dealing with mobility and Petri nets can be found in [Val98], [Bus99], [Lom00], [XD00], [Hir02], [KMR03], [BBPP04], [Lak05], and [KF06].

Continuing previous work [KMR03, KF06] we study object-based Petri nets, especially, to model mobile agents. In this paper we generalise the established formalism of *elementary object system* (EOS) [Val98] to cover requirements of mobile systems. The formalism we introduce here is called *mobile, elementary object system* (mobile EOS).

Valk introduced two fundamental semantics (i.e. firing rules) for EOS called *reference semantics* and *value semantics* (cf. [Val98]) which correspond to the two ways of handling objects in Java-like programming languages: Reference semantics accesses object nets using pointers, while value semantics is a kind of copy semantics.

The difference between the two is the concept of *location* for net tokens which is explicit for value semantics but not for reference semantics, since it is unclear which reference can be considered as the location of a net token. As shown in [KMR03] the concept of mobility cannot be expressed adequately by reference semantics due to the possibility of side-effects. Reference semantics considers all places as one single name space. In Figure 1, when the agent executes (t_5, t_{22}) at building B, then the effect becomes visible immediately at building A. This is undesired, since it should become visible only after the agent returns to A.

For value semantics each place is its own name space, so it cannot simulate the global name space[1] as demanded by our scenario for the access inside buildings. When firing t_1 two net tokens with independent marking are created. When firing (t_2, t_{20}) the effect is only visible in the copy moving from p_3 to p_4, but not for the net token in p_2. Since p_2 and p_4 are in different name spaces for the normal firing semantics the effect is not visible at p_2 and (t_3, t_{21}) never becomes enabled.

Mobile EOS overcome the shortcomings of EOS by introducing modelling constructs that allow us to define name spaces that are larger than one place (as value semantics does) but do not necessarily comprise of the whole net (as reference semantics does). The fact that mobile EOS subsume value and reference semantics is reflected by the fact that they are strictly more expressive than EOS, since they are able to simulate inihibitor nets.

The paper is structured as follows. Section 2 defines elementary object systems. In Section 3 we extend these to mobile EOS. In Section 4 we prove some properties of mobile EOS including the simulation of inhibitor nets. Section 5 introduces object nets where the system net is clustered into locations and we demonstrate how mobile EOS are used to implement name spaces.

2 Object Nets

Object nets generalise place/transition nets. We recall basic notations.

Multisets. The definition of Petri nets relies on the notion of multisets. A multiset $\mathbf{m}$ on the set D is a mapping $\mathbf{m} : D \to \mathbb{N}$. Multisets are generalisations of sets in

[1] The concept of locality makes value semantics richer than reference semantics – for example the reachability problem becomes undecidable while boundedness remains decidable (cf. [KR04]). However, reference semantics can be simulated by a (larger) P/T net, so analysis methods can be applied directly.

the sense that every subset of D corresponds to a multiset $\mathbf{m}$ with $\mathbf{m}(d) \leq 1$ for all $d \in D$. The notation is overloaded, being used for sets as well as for multisets. The meaning will be apparent from its use.

Multiset addition $\mathbf{m}_1, \mathbf{m}_2 : D \to \mathbb{N}$ is defined component-wise: $(\mathbf{m}_1+\mathbf{m}_2)(d) := \mathbf{m}_1(d)+\mathbf{m}_2(d)$. The empty multiset $\mathbf{0}$ is defined as $\mathbf{0}(d) = 0$ for all $d \in D$. Multiset-difference $\mathbf{m}_1 - \mathbf{m}_2$ is defined $(\mathbf{m}_1 - \mathbf{m}_2)(d) := \max(\mathbf{m}_1(d) - \mathbf{m}_2(d), 0)$. We use common notations for the cardinality of a multiset $|\mathbf{m}| := \sum_{d \in D} \mathbf{m}(d)$ and multiset ordering $\mathbf{m}_1 \leq \mathbf{m}_2$ where the partial order $\leq$ is defined by $\mathbf{m}_1 \leq \mathbf{m}_2 \iff \forall d \in D : \mathbf{m}_1(d) \leq \mathbf{m}_2(d)$.

A multiset $\mathbf{m}$ is finite if $|\mathbf{m}| < \infty$. The set of all finite multisets over the set D is denoted $MS(D)$, so we also denote $\mathbf{m} \in MS(D)$. The set of all finite multisets $MS(D)$ over D naturally forms a monoid with multiset addition $+$ and the empty multiset $\mathbf{0}$. Multisets can be identified with the commutative monoid structure $(MS(D), +, 0)$.

Multisets are the free commutative monoid over D since every multiset has the unique representation in the form $\mathbf{m} = \sum_{d \in D} \mathbf{m}(d) \cdot d$ where $\mathbf{m}(d)$ denotes the multiplicity of d. Multisets can be represented as a formal sum in the form $\mathbf{m} = \sum_{i=1}^{|\mathbf{m}|} x_i$ where $x_i \in D$.

Any mapping $f : D \to D'$ can be extended to a mapping $f^\sharp : MS(D) \to MS(D')$ on multisets in a linear way: $f^\sharp \left(\sum_{i=1}^{n} x_i\right) = \sum_{i=1}^{n} f(x_i)$. This includes the special case $f^\sharp(\mathbf{0}) = \mathbf{0}$. We simply write f to denote the mapping $f^\sharp$. The definition is in accordance with the set-theoretic notation $f(A) = \{f(a) \mid a \in A\}$.

P/T Nets Place/transition nets (P/T nets) are Petri nets with arc weights, expressed by multisets, and the possibility of multiple tokens on each place.

Definition 1. *A* P/T net N *is a tuple* $N = (P, T, \mathbf{pre}, \mathbf{post})$, *such that:*

1. P *is a set of places.*
2. T *is a set of transitions, with* $P \cap T = \emptyset$.
3. $\mathbf{pre}, \mathbf{post} : T \to MS(P)$ *are the pre- and post-condition functions.*

A marking of N *is a multiset of places:* $\mathbf{m} \in MS(P)$. *A P/T net with initial marking* $\mathbf{m}$ *is denoted* $N = (P, T, \mathbf{pre}, \mathbf{post}, \mathbf{m})$.

We use the usual notations for nets like ${}^\bullet x$ for the set of predecessors and $x^\bullet$ for the set of successors for a node $x \in (P \cup T)$.

A transition $t \in T$ of a P/T net N is enabled in marking $\mathbf{m}$ iff $\forall p \in P : \mathbf{m}(p) \geq \mathbf{pre}(t)(p)$ holds. The successor marking when firing t is $\mathbf{m}'(p) = \mathbf{m}(p) - \mathbf{pre}(t)(p) + \mathbf{post}(t)(p)$. We denote the enabling of t in marking $\mathbf{m}$ by $\mathbf{m} \xrightarrow[N]{t}$. Firing of t is denoted by $\mathbf{m} \xrightarrow[N]{t} \mathbf{m}'$. The net N is omitted if it is clear from the context.

2.1 Elementary Object Systems

Object nets are Petri nets which have Petri nets as tokens – an approach called the *nets-within-nets* paradigm, proposed by Valk [Val91, Val03] for a two levelled structure and generalised in [KR03, KR04] for arbitrary nesting structures.

Objects nets are useful to model the mobility of active objects or agents (cf. [KMR01] and [KMR03]).

Figure 2 shows an example object net. The Petri nets that are used as tokens are called *net tokens*. Net tokens are tokens with internal structure and inner activity. The *nets as tokens* perspective is different from place refinement, since tokens are transported while a place refinement is static. Net tokens can be viewed as a kind of *dynamic refinement* of states.

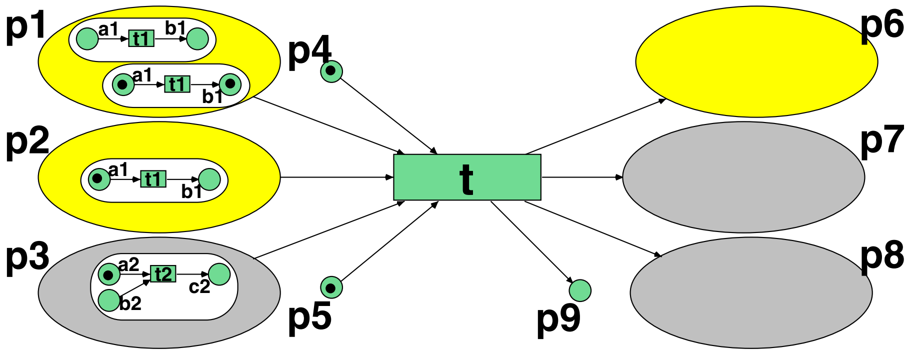


Fig. 2. An Object Net

An elementary object system (EOS) is composed of a system net, which is a P/T net $\widehat{N} = (\widehat{P}, \widehat{T}, \mathbf{pre}, \mathbf{post})$ and a finite set of object nets $\mathcal{N} = \{N_1, \ldots, N_n\}$, which are P/T nets given as $N = (P_N, T_N, \mathbf{pre}_N, \mathbf{post}_N)$. Each $N \in \mathcal{N}$ models a different mobile agent in the system. In the example in Figure 1 we have the special case that $|\mathcal{N}| = 1$, i.e. there is only one agent in the system.

Without loss of generalisation we assume that all sets of nodes (places and transitions) are disjoint. So, $\widehat{P} \cup \bigcup_{N \in \mathcal{N}} P_N$ is a disjoint union (cf. Definition 7).

The system net places are typed by the mapping $d : \widehat{P} \to \{\bullet\} \cup \mathcal{N}$ with the meaning that a place $\widehat{p}$ of the system net may contain only black tokens if $d(\widehat{p}) = \bullet$ and only net tokens of the object net type N if $d(\widehat{p}) = N$. The set of system net places of the type N is $d^{-1}(N) \subseteq \widehat{P}$.

The typing d is called *monotonous* iff $\forall t \in T : \forall N \in \mathcal{N} : N \in d({}^\bullet\widehat{t}) \Longrightarrow N \in d(\widehat{t}^\bullet)$. We restrict EOS to monotonous mappings to ensure monotonicity of the firing rule (see Theorem 4 below).

Markings. Each net token in an EOS is an instances of an object net. Usually, we have several, independent net tokens derived from the same object net. The net tokens usually have different markings. A *marking* $\mu \in \mathcal{M}_\mathcal{N}$ of an EOS *OS* is a nested multiset where $\mathcal{M}_\mathcal{N}$ is the set of all possible markings, defined in the following way:

$$\mathcal{M}_\mathcal{N} := MS\left(\left(d^{-1}(\bullet) \times \{\mathbf{0}\}\right) \cup \bigcup_{N \in \mathcal{N}} \left(d^{-1}(N) \times MS(P_N)\right)\right) \tag{1}$$

A marking of an Eos OS is denoted $\mu = \sum_{k=1}^{|\mu|}(\widehat{p}_k, M_k)$ where $\widehat{p}_k$ is a place in the system net and M_k is the marking of the net token of type $d(\widehat{p}_k)$.

Let $\mu = \sum_{k=1}^{|\mu|}(\widehat{p}_k, M_k)$ be a marking of an Eos OS. By using projections on the first or last component of an Eos marking μ, it is possible to compare object system markings. The projection $\Pi^1(\mu)$ on the first component abstracts from the substructure of a net token:

$$\Pi^1\left(\sum\nolimits_{k=1}^{|\mu|}(\widehat{p}_k, M_k)\right) := \sum\nolimits_{k=1}^{|\mu|}\widehat{p}_k \tag{2}$$

The projection $\Pi_N^2(\mu)$ on the second component is the abstract marking of all the net tokens of type $N \in \mathcal{N}$ without considering their local distribution in the system net.

$$\Pi_N^2\left(\sum\nolimits_{k=1}^{|\mu|}(\widehat{p}_k, M_k)\right) := \sum\nolimits_{k=1}^{|\mu|}\pi_N(\widehat{p}_k) \cdot M_k \tag{3}$$

where the indicator function $\pi_N : \widehat{P} \to \{0, 1\}$ is $\pi_N(\widehat{p}) = 1$ iff $d(\widehat{p}) = N$.

Events. The Eos firing rule is defined for three cases: system-autonomous firing, object-autonomous firing, and synchronised firing. For the sake of uniformity of the firing rule, for each $N \in \mathcal{N}$, we add the idle transitions ϵ_N for the object net where $\mathbf{pre}(\epsilon_N) = \mathbf{post}(\epsilon_N) = \mathbf{0}$ and the set of idle transitions $\{\epsilon_{\widehat{p}} \mid \widehat{p} \in \widehat{P}\}$ where $\mathbf{pre}(\epsilon_{\widehat{p}}) = \mathbf{post}(\epsilon_{\widehat{p}}) = \widehat{p}$ for the system net.

Let $\mathcal{C}$ be the set of all mappings $C : \mathcal{N} \to \bigcup_{N\in\mathcal{N}}(T_N \cup \{\epsilon_N\})$ such that each $C \in \mathcal{C}$ maps each object net $N \in \mathcal{N}$ to one of its transitions $t \in T_N$ or the idle transition ϵ_N, i.e. $C(N) \in (T_N \cup \{\epsilon_N\})$. The idle map $\epsilon_{\mathcal{C}} \in \mathcal{C}$ is defined $\epsilon_{\mathcal{C}}(N) = \epsilon_N$ for all $N \in \mathcal{N}$. For $\mathcal{N} = \{N_1, \ldots, N_n\}$ we denote $C \in \mathcal{C}$ as the tuple $(C(N_1), \ldots, C(N_n))$.

An event of the whole system is a pair $(\widehat{\tau}, C)$ where $\widehat{\tau}$ is a transition of the system net or $\epsilon_{\widehat{p}}$ and $C \in \mathcal{C}$:

$$\mathcal{T} = \left\{(\widehat{\tau}, C) \,\middle|\, \widehat{\tau} \in \widehat{T} \cup \{\epsilon_{\widehat{p}} \mid \widehat{p} \in \widehat{P}\} \wedge C \in \mathcal{C}\right\} \setminus \left\{(\epsilon_{\widehat{p}}, \epsilon_{\mathcal{C}}) \,\middle|\, \widehat{p} \in \widehat{P}\right\} \tag{4}$$

An event $(\widehat{\tau}, C)$ has the meaning that $\widehat{\tau}$ fires synchronously with all the object net transitions $C(N)$ for $N \in \mathcal{N}$. Note, that $(\epsilon_{\widehat{p}}, \epsilon_{\mathcal{C}})$ is excluded because it has no effect. By the construction of $\mathcal{T}$ each system net transition has exactly one synchronisation partner in the object net $N \in \mathcal{N}$. This partner might be an idle-transition. System-autonomous events have the form $(\widehat{t}, \epsilon_{\mathcal{C}})$. For a single object-autonomous event at the location $\widehat{p}$ we have $\widehat{\tau} = \epsilon_{\widehat{p}}$ and for all except one object net N we have $C(N) = \epsilon_N$, i.e. $|C(\mathcal{N}) \setminus \epsilon_{\mathcal{C}}| = 1$.

A set $\Theta \subseteq \mathcal{T}$ is called a *synchronisation structure*. In the graphical representation of object nets, synchronisation structures are defined by transition labels of the form $\langle$: name() $\rangle$ which means that a labelled system net transition has to be synchronised with an object net transition with the same label (where labels of the form ϵ_N and $\epsilon_{\widehat{p}}$ are ommitted).

Definition 2. *An* Eos *is a tuple* $OS = (\widehat{N}, \mathcal{N}, d, \Theta, \mu_0)$ *such that:*

1. $\widehat{N}$ *is a P/T net, called the* system net.
2. $\mathcal{N}$ *is a finite set of P/T nets, called* object nets.
3. $d : \widehat{P} \to \{\bullet\} \cup \mathcal{N}$ *is a monotonous typing of the system net places.*
4. $\Theta \subseteq \mathcal{T}$ *is a finite synchronisation structure.*
5. $\mu_0 \in \mathcal{M}_{\mathcal{N}}$ *is the initial marking.*

We name special properties of Eos*:*

- *An* Eos *is* minimal *iff it has exactly one type of object nets:* $|\mathcal{N}| = 1$.
- *An* Eos *is* pure *iff it has no places for black tokens:* $d^{-1}(\bullet) = \emptyset$.
- *An* Eos *is* p/t-like *iff it has only places for black tokens:* $d(\widehat{P}) = \{\bullet\}$.
- *An* Eos *is* unary *iff it is pure and minimal.*

Example 1. Figure 2 shows an Eos with the system net $\widehat{N}$ and the object nets $\mathcal{N} = \{N_1, N_2\}$. The nets are given as $\widehat{N} = (\widehat{P}, \widehat{T}, \mathbf{pre}, \mathbf{post})$ with $\widehat{P} = \{p_1, \ldots, p_9\}$ and $\widehat{T} = \{t\}$; $N_1 = (P_1, T_1, \mathbf{pre}_1, \mathbf{post}_1)$ with $P_1 = \{a_1, b_1\}$ and $T_1 = \{t_1\}$; and $N_2 = (P_2, T_2, \mathbf{pre}_2, \mathbf{post}_2)$ with $P_2 = \{a_2, b_2, c_2\}$ and $T_2 = \{t_2\}$.
The typing is $d(p_1) = d(p_2) = d(p_6) = N_1$, $d(p_3) = d(p_7) = d(p_8) = N_2$, and $d(p_4) = d(p_5) = d(p_9) = \bullet$. The typing is illustrated in Figure 2 by different colours for the places. There are only autonomous events:

$$\Theta = \{(t, (\epsilon_{N_1}, \epsilon_{N_2}))\} \cup \{(\epsilon_{\widehat{p}}, (t_1, \epsilon_{N_2})), (\epsilon_{\widehat{p}}, (\epsilon_{N_1}, t_2)) \mid \widehat{p} \in \widehat{P}\}$$

The initial marking has black tokens in p_4 and p_5, two net tokens in p_1, and one net token each in p_2 and p_3:

$$\mu = (p_1, \mathbf{0}) + (p_1, a_1 + b_1) + (p_2, a_1) + (p_3, a_2 + b_2) + (p_4, \mathbf{0}) + (p_5, \mathbf{0})$$

Note, that for Figure 2 the structure is the same for the three net tokens in p_1 and p_2 but the net tokens' markings are different.

We have to consider three different kinds of events: system-autonomous firing, object-autonomous firing, and synchronised firing. Due to the idle step we have a uniform structure of events $(\widehat{\tau}, C) \in \mathcal{T}$ and the conditions on λ and ρ for the different kinds of firing are expressed by the *enabling predicate* ϕ:

$$\begin{aligned}\phi((\widehat{\tau}, C), \lambda, \rho) \iff & \Pi^1(\lambda) = \mathbf{pre}(\widehat{\tau})) \wedge \Pi^1(\rho) = \mathbf{post}(\widehat{\tau}) \wedge \\ & \forall N \in \mathcal{N} : \Pi^2_N(\lambda) \geq \mathbf{pre}_N(C(N)) \wedge \\ & \forall N \in \mathcal{N} : \Pi^2_N(\rho) = \Pi^2_N(\lambda) - \mathbf{pre}_N(C(N)) + \mathbf{post}_N(C(N))\end{aligned}$$

1. The first conjunct expresses that the first component of the nested multiset λ corresponds to the pre-condition of the system net transition $\widehat{\tau}$: $\Pi^1(\lambda) = \mathbf{pre}(\widehat{\tau})$.
2. In turn, a nested multiset $\rho \in \mathcal{M}_{\mathcal{N}}$ is produced, that corresponds with the post-set of $\widehat{\tau}$ in its first component.

3. An object net transition τ_N is enabled if the combination of the net tokens of type N enable it, i.e. $\Pi^2_N(\lambda) \geq \mathbf{pre}_N(C(N))$.
4. The firing of $\widehat{\tau}$ must also obey the *object marking distribution condition* $\Pi^2_N(\lambda) = \Pi^2_N(\rho) - \mathbf{pre}_N(C(N)) + \mathbf{post}_N(C(N))$ where $\mathbf{post}_N(C(N)) - \mathbf{pre}_N(C(N))$ is the effect of the object net's transition on the net tokens.

For system-autonomous events $(\widehat{t}, \epsilon_{\mathcal{C}})$ the enabling predicate ϕ can be simplified further. We have $\mathbf{pre}_N(\epsilon_N) = \mathbf{post}_N(\epsilon_N) = \mathbf{0}$. This ensures that $\Pi^2_N(\lambda) = \Pi^2_N(\rho)$, i.e. the sum of markings in the copies of a net token is preserved with respect to each type N. This condition ensures the existence of linear invariance properties (cf. [KR04]). Analogously, for an object-autonomous event we have the idle transition $\widehat{\tau} = \epsilon_{\widehat{p}}$ for the system net and the first and the second conjunct in ϕ is: $\Pi^1(\lambda) = \mathbf{pre}(\widehat{\tau}) = \widehat{p} = \mathbf{post}(\widehat{\tau}) = \Pi^1(\rho)$. So, there is an addend $\lambda = (\widehat{p}, M)$ with $d(\widehat{p}) = N$ and the marking M enables the object net transition.

The predicate ϕ does not distinguish between markings that coincide in their projections. To express this fact we define the equivalence $\cong\ \subseteq \mathcal{M}^2_{\mathcal{N}}$ that relates nested markings which coincide in their projections:

$$\alpha \cong \beta :\iff \Pi^1(\alpha) = \Pi^1(\beta) \wedge \forall N \in \mathcal{N} : \Pi^2_N(\alpha) = \Pi^2_N(\beta) \qquad (5)$$

The relation $\alpha \cong \beta$ abstracts from the location, i.e. the concrete net-token, in which a object net's place p is marked as long as it is present in α and β. For example, we have

$$(\widehat{p}, p_1 + p_2) + (\widehat{p}', p_3) \cong (\widehat{p}, p_3 + p_2) + (\widehat{p}', p_1)$$

which means that $\cong$ allows the tokens p_1 and p_3 to change their locations (i.e. $\widehat{p}$ and $\widehat{p}'$). It is not allowed that the token itself is modified, i.e. the token p_1 cannot change into p_1'.

Lemma 1. *The enabling predicate is insensitive with respect to the relation* $\cong$*:*

$$\phi((\widehat{\tau}, C), \lambda, \rho) \iff \forall \lambda', \rho' : \lambda' \cong \lambda \wedge \rho' \cong \rho \Longrightarrow \phi((\widehat{\tau}, C), \lambda', \rho')$$

Proof. From the definition of ϕ one can see that the firing mode (λ, ρ) is used only via the projections $\Pi^{1,2}$. Since $\lambda' \cong \lambda, \rho' \cong \rho$ expresses equality modulo projection ϕ cannot distinguish between λ' and λ, resp. ρ' and ρ.

Firing an event $(\widehat{\tau}, C)$ involving the system net transition $\widehat{\tau}$ removes net tokens in the pre-conditions together with their individual internal markings. Since the markings of Eos are nested multisets, we have to consider those nested multisets $\lambda \in \mathcal{M}_{\mathcal{N}}$ that are part of the current marking, i.e. $\lambda \leq \mu$.

For the definition of firing we use the projection equivalence to express that on firing the system net collects all relevant object nets for the particular firing mode and combines them into one virtual object net that is only present at the moment of firing. Due to this collection the location of the object nets' tokens is irrelevant and can be ignored using projection equivalence.

Definition 3. *Let OS be an* EOS *and let $\mu, \mu' \in \mathcal{M}_{\mathcal{N}}$ be markings. The transition $(\widehat{\tau}, C) \in \mathcal{T}$ is* enabled *in mode $(\lambda, \rho) \in \mathcal{M}_{\mathcal{N}}^2$ if the following holds:*

$$\lambda \leq \mu \wedge \exists \lambda', \rho' : (\lambda' \cong \lambda) \wedge (\rho' \cong \rho) \wedge \phi((\widehat{\tau}, C), \lambda', \rho')$$

The successor marking is defined as $\mu' := \mu - \lambda + \rho$.

We write $\mu \xrightarrow[OS]{(\widehat{\tau},C)} \mu'$ whenever $\mu \xrightarrow[OS]{(\widehat{\tau},C)(\lambda,\rho)} \mu'$ for some mode (λ, ρ).

From Lemma 1 and Def. 3 we conclude that $\lambda \xrightarrow[OS]{(\widehat{\tau},C)(\lambda,\rho)} \rho$ iff we have for all λ', ρ' with $\lambda' \cong \lambda$ and $\rho' \cong \rho$ that $\lambda' \xrightarrow[OS]{(\widehat{\tau},C)(\lambda',\rho')} \rho'$. Hence, we see that the enabling condition given in Definition 3 can be strengthened:

$$\mu \xrightarrow[OS]{(\widehat{\tau},C)(\lambda,\rho)} \mu' \iff (\lambda \leq \mu \wedge \phi((\widehat{\tau}, C), \lambda, \rho)) \tag{6}$$

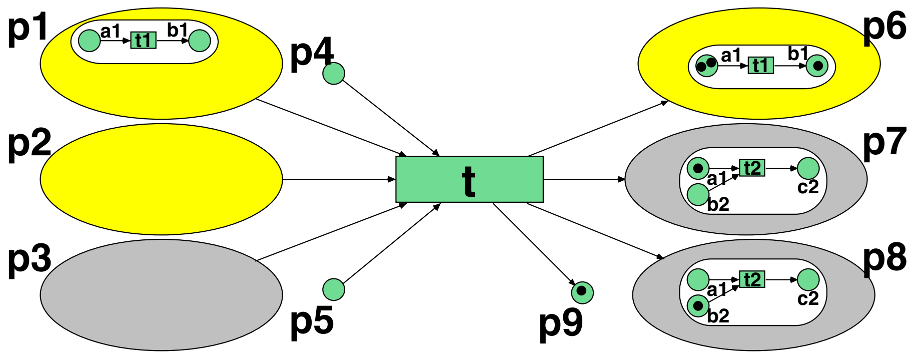


Fig. 3. The EOS of Figure 2 after the firing of $(\widehat{t}, (\epsilon_{N_1}, \epsilon_{N_2}))$

Example 2. Consider the EOS of Figure 2 again. The event $(\widehat{t}, (\epsilon_{N_1}, \epsilon_{N_2}))$ is enabled in mode $(\lambda, \rho) \in \mathcal{M}_{\mathcal{N}}^2$ with

$$\begin{aligned} \lambda &= (p_1, a_1 + b_1) + (p_2, a_1) + (p_3, a_2 + b_2) + (p_4, \mathbf{0}) + (p_5, \mathbf{0}) \\ \rho &= (p_6, a_1 + a_1 + b_1) + (p_7, a_2) + (p_8, b_2) + (p_9, \mathbf{0}) \end{aligned}$$

After firing in this mode we obtain the successor marking (cf. Figure 3):

$$\mu' = (p_1, \mathbf{0}) + (p_6, a_1 + a_1 + b_1) + (p_7, a_2) + (p_8, b_2) + (p_9, \mathbf{0})$$

A transition $\widehat{t} \in \widehat{T}$ with an an object net N that is present in the postset but not in the preset (i.e. $N \notin d({}^\bullet\widehat{t})$ and $N \in d(\widehat{t^\bullet})$) generates net tokens of type N. The firing rule ensures that these net tokens carry the empty marking since in this case $(\widehat{\tau}, C)$ is enabled in mode (λ, ρ) only if all object nets in ρ of this type N carry the empty marking.

The converse case, i.e. $N \in d({}^\bullet\widehat{t})$ and $N \notin d(\widehat{t}^\bullet)$, which destroys net tokens of type N, is forbidden by the monotonous typing, since it yields a contradiction: In this case $(\widehat{\tau}, C)$ is enabled in mode (λ, ρ) only if all object nets in λ of this type N carry the empty marking: $\Pi_N^2(\lambda) = \mathbf{0}$. Hence, not all pairs (λ', ρ') with $\lambda \preceq \lambda'$ are also firing modes, i.e. the firing rule would *not* be monotonous.

2.2 Name Spaces for EOS

With Eos we can define a system net like that in Figure 1, but the net structures for the buildings do not have the intended meaning. As discussed in the introduction, the firing rule for Eos treats each place as a singular name space.

In the following we define a mobility system with more flexible name spaces. Given an Eos the system net N is decomposed into locations (the name spaces) and mobility parts. A *location* is a subnet $L = (P_L, T_L, \mathbf{pre}_L, \mathbf{post}_L)$ of the system net N, i.e. $P_L \subseteq P$, $T_L \subseteq T$, $\mathbf{pre}_L = \mathbf{pre}|_{T_L,P_L}$, and $\mathbf{post}_L = \mathbf{post}|_{T_L,P_L}$. Location nets are disjoint.

The location nets are connected by transitions $T_m \subseteq T$ that describe the movement from one location to another.

Definition 4. *A locality infrastructure is the tuple $LS = (\mathcal{L}, N_m)$ where:*

1. *$\mathcal{L}$ is a finite set of disjoint nets, called* locations*, given as $L = (P_L, T_L, \mathbf{pre}_L, \mathbf{post}_L)$.*
2. *$N_m = (P_m, T_m, \mathbf{pre}_m, \mathbf{pre}_m)$ is the mobility infrastructure disjoint from all $L \in \mathcal{L}$ where $P_m = \emptyset$ and for all $t_m \in T_m$ exist two location nets $L, L' \in \mathcal{L}$ with $L \neq L'$ that are connected by t_m:*

$$\{p \in P \mid \mathbf{pre}_m(t_m)(p) > 0\} \subseteq P_L \wedge \{p \in P \mid \mathbf{post}_m(t_m)(p) > 0\} \subseteq P_{L'}$$

The net $N(LS) := (P, T, \mathbf{pre}, \mathbf{post})$ generated by an infrastructure LS is given by $P = \bigcup_{L \in \mathcal{L}} P_L$, $T = T_m \cup \bigcup_{L \in \mathcal{L}} T_L$, $\mathbf{pre} = \mathbf{pre}_m \cup \bigcup_{L \in \mathcal{L}} \mathbf{pre}_L$, and $\mathbf{post} = \mathbf{post}_m \cup \bigcup_{L \in \mathcal{L}} \mathbf{post}_L$.

A mobility system *is the pair (OS, LS) where $OS = (\widehat{N}, \mathcal{N}, d, \Theta, \mu_0)$ is an* Eos *and $LS = (\mathcal{L}, N_m)$ is a locality infrastructure generating the system net: $N(LS) = \widehat{N}$.*

Example 3. The Eos in Figure 1 has two locations A and B indicated by the buildings' borderlines: $P_A = \{p_1, p_2, p_3, p_4, p_5, p_8, p_9, p_{10}\}$, $T_A = \{t_1, t_2, t_3, t_7, t_8\}$ and $P_B = \{p_6, p_6\}$ $T_B = \{t_5\}$. The mobility transitions are $T_m = \{t_4, t_6\}$.

3 Mobile EOS

A first attempt at a solution to redefine the firing rule was to make it respect the structure of the mobility system (OS, LS). Following this approach we changed the firing rule to allow local and global access. However, the approach turns out to be unnecessarily specific to the domain of name spaces. Instead we extend the formalism in a more general way. (Section 5 describes how these extensions are actually used to model name spaces).

So far the firing rule was insensitive with respect to the relation $\cong$ which ignores the system net location of an object net's token but requires that the total number of tokens in different copies of the same object net place adds up to the number expected when firing the respective transition in a traditional net. We generalise this notion by using a more general relation.

For Eos each place is its own name space. To define clusters of places belonging to the same name space we introduce conversion sets: An equivalence $\leftrightarrow$ on the object net's places $P = \bigcup_{N \in \mathcal{N}} P_N$ is called a *conversion*, its equivalence classes are *conversion sets*. Let $P/_{\leftrightarrow}$ denote the set of equivalence classes of $\leftrightarrow$ and let $g_{\leftrightarrow} : P \to P/_{\leftrightarrow}$ be the natural surjection of $P/_{\leftrightarrow}$, i.e. the function that maps each place $p \in P$ to the conversion set that p belongs to: $g_{\leftrightarrow}(p) = \{p' \mid p' \leftrightarrow p\}$. For the identity conversion $\leftrightarrow\; = id_P$ we obtain $g_{\leftrightarrow}$ as the injection: $g_{id_P}(p) = \{p\}$.

Definition 5. *A* mobile EOS *is a pair* $(OS, \leftrightarrow)$ *where* OS *is an* Eos *and* $\leftrightarrow$ *is a conversion equivalence on the object nets' places* $P = \bigcup_{N \in \mathcal{N}} P_N$*.*

We extend $\leftrightarrow$ to an equivalence on nested multisets. The equivalence $\leftrightarrow$ identifies $\alpha, \beta \in \mathcal{M}_{\mathcal{N}}$ whenever the sum of tokens in all net tokens of type N (which is $\Pi^2_N(\alpha)$ and $\Pi^2_N(\beta)$) is equal in both nested multisets modulo $\leftrightarrow$:

$$\alpha \leftrightarrow \beta \iff \Pi^1(\alpha) = \Pi^1(\beta) \wedge \forall N \in \mathcal{N} : g^\sharp_{\leftrightarrow}(\Pi^2_N(\alpha)) = g^\sharp_{\leftrightarrow}(\Pi^2_N(\beta)) \qquad (7)$$

The relation $\alpha \leftrightarrow \beta$ abstracts from the concrete location $\widehat{p}$ in the system net in which an object net place p is marked, and additionally allows token conversions via $\leftrightarrow$.

Lemma 2. *For all* $\alpha, \beta \in \mathcal{M}$ *and all coversions* $\leftrightarrow$ *we have* $\alpha \cong \beta \Longrightarrow \alpha \leftrightarrow \beta$*. In particular, for* $\leftrightarrow = id_P$ *we have* $\alpha \cong \beta \iff \alpha \leftrightarrow \beta$*.*

Proof. Directly from (5) and (7).

We modify the firing rule given in Definition 3 using the equivalence $\leftrightarrow$ instead of $\cong$.

Definition 6. *Let* $(OS, \leftrightarrow)$ *be a mobile* EOS *and let* $\mu, \mu' \in \mathcal{M}_{\mathcal{N}}$ *be markings. The transition* $(\widehat{\tau}, C) \in \mathcal{T}$ *is* $\leftrightarrow$-enabled *in mode* $(\lambda, \rho) \in \mathcal{M}^2_{\mathcal{N}}$ *if the following holds:*

$$\lambda \leq \mu \wedge \exists \lambda', \rho' : (\lambda' \leftrightarrow \lambda) \wedge (\rho' \leftrightarrow \rho) \wedge \phi((\widehat{\tau}, C), \lambda', \rho')$$

The successor marking is defined as $\mu' := \mu - \lambda + \rho$*.*

4 Relating Mobile EOS to EOS

There is an obvious construction of a P/T net, called the reference net, which is constructed by taking as the set of places the disjoint union of all places and as the set of transitions the synchronisations.

Definition 7. *Let $OS = (\widehat{N}, \mathcal{N}, d, \Theta, \mu_0)$ be an* EOS. *The* reference net $\text{RN}(OS)$ *is defined as the P/T net:*

$$\text{RN}(OS) = \left(\left(\widehat{P} \cup \bigcup\nolimits_{N \in \mathcal{N}} P_N\right), \Theta, \mathbf{pre}^{\text{RN}}, \mathbf{post}^{\text{RN}}, \text{RN}(\mu_0)\right)$$

where $\mathbf{pre}^{\text{RN}}$ *(and analogously* $\mathbf{post}^{\text{RN}}$*) is defined by:*

$$\mathbf{pre}^{\text{RN}}((\widehat{\tau}, C)) = \mathbf{pre}(\widehat{\tau}) + \sum\nolimits_{N \in \mathcal{N}} \mathbf{pre}_N(C(N))$$

and for markings we define:

$$\text{RN}\left(\sum\nolimits_{k=1}^{|\mu|} (\widehat{p}_k, M_k)\right) = \sum\nolimits_{k=1}^{|\mu|} \widehat{p}_k + M_k$$

The net is called reference net because it behaves as if each object net was accessed via pointers and not like a value. We have the following property [KR04, Proposition 1]:

Theorem 1. *Let OS be an* EOS. *Every transition $(\widehat{\tau}, C) \in \mathcal{T}$ that is activated in OS for (λ, ρ) is so in $\text{RN}(OS)$:*

$$\mu \xrightarrow[OS]{(\widehat{\tau},C)} \mu' \Longrightarrow \text{RN}(\mu) \xrightarrow[\text{RN}(OS)]{(\widehat{\tau},C)} \text{RN}(\mu')$$

If we use $\text{RN}(OS)$ as the only object net (i.e. $\mathcal{N} = \{\text{RN}(OS)\}$) and a system net that has one single place only, then this EOS simulates the reference semantics. So, we have shown that for EOS the reference semantics is a special case of the (value) semantics. As another property we obtain, that the definition of $\leftrightarrow$-enabling is a canonical extension of the value semantics given in [KR04]:

Theorem 2. *The mobile* EOS (OS, id_P) *has the same behaviour as the* EOS *OS: The transition $(\widehat{\tau}, C) \in \mathcal{T}$ is enabled in mode (λ, ρ) for OS iff it is id_P-enabled in mode (λ, ρ) for $(OS, \leftrightarrow)$.*

Proof. From Lemma 2 we have $\alpha \cong \beta \iff \alpha \leftrightarrow \beta$ whenever $\leftrightarrow\ = id_P$. So, Definition 6 simplifies to Definition 3 and each transition $(\widehat{\tau}, C) \in \mathcal{T}$ is enabled in mode (λ, ρ) for value semantics iff it is id_P-enabled in the mobile EOS.

The special case of $\leftrightarrow\ = id_P$ expresses the fact that on firing, the system net collects all relevant object nets for the particular firing mode and combines them into one virtual object net that is only present at the moment of firing. Due to this collection, the location of the object net's tokens is irrelevant and is ignored.

For EOS reachability is undecidable [Köh06, Theorem 2].

Theorem 3. *Reachability is undecidable for non-minimal, pure* EOS *and for minimal, non-pure* EOS.

For EOS boundedness remains decidable [Köh06, Theorem 7].

Theorem 4. *Boundedness is decidable for* Eos.

The following theorem shows that mobile Eos are more powerful than Eos, because neither reachability nor boundedness nor coverability are decidable for inhibitor nets while boundedness is decidable for Eos.

Theorem 5. *Mobile* Eos *can simulate Petri nets with inhibitor arcs.*

We recall the definition of inhibitor nets:

Definition 8. *An* inhibitor net *is a tuple* (N, A) *where* N *is a P/T net and* $A \subseteq (P \times T)$ *is a set of inhibitor arcs.*

A transition $t \in T$ *is enabled in marking* $\mathbf{m}$ *(denoted* $\mathbf{m} \xrightarrow[(N,A)]{t}$*) iff* $\mathbf{m} \xrightarrow[N]{t}$ *and* $\mathbf{m}(p) = 0$ *for all* $p \in ({}_{\_}At)$*. The successor marking is* $\mathbf{m}'(p) = \mathbf{m}(p) - \mathbf{pre}(t)(p) + \mathbf{post}(t)(p)$ *for all* $p \in P$*.*

A place $p \in ({}_{\_}At)$ is called an *inhibitor place* of t. For an inhibitor net a transition t is disabled whenever an inhibitor place is marked. If $A = \emptyset$ then an inhibitor net (N, A) behaves as the P/T net N itself.

In the following we define for an arbitrary inhibitor net a mobile Eos that simulates it. This is sufficient to prove Theorem 5.

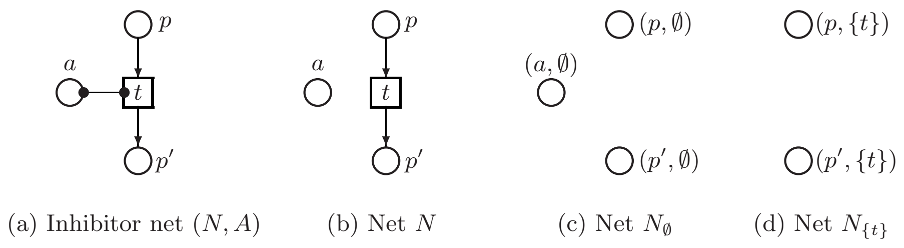


Fig. 4. The inhibitor net (N, A) and its subnets

Given an inhibitor net (N, A) we define $\mathcal{Z} := \{\emptyset\} \cup \{\{t\} \mid t \in inh(T)\}$ where $inh(T)$ is the subset of transitions with inhibitor arcs: $inh(T) := (PA_{\_})$. For the inhibitor net N in Figure 4 (a) we have $\mathcal{Z} = \{\emptyset, \{t\}\}$.

For each $Z \in \mathcal{Z}$ we define the P/T net

$$N_Z := ((P \setminus ({}_{\_}AZ)) \times \{Z\}, \emptyset, \emptyset, \emptyset)$$

which is obtained by dropping all transitions and the places that are tested for emptiness by the transition in Z. To make all the nets N_Z disjoint we use Z as the second component for the places. The nets $N_\emptyset$ and $N_{\{t\}}$ are shown in Figures 4 (c) and (d).

Next we construct a mobile Eos $Inh(N, A)$ for the inhibitor net (N, A). We use $\mathcal{N} = \{N_Z \mid Z \in \mathcal{Z}\}$ as the set of object nets. The system net contains the place $\widehat{p}_0$ that carries net tokens of type N, i.e. $d(\widehat{p}_0) = N$, and one place $\widehat{p}^Z$ for each $Z \in \mathcal{Z}$ with $d(\widehat{p}^Z) = N_Z$. For each transition $t \in T$ we add the transitions $\widehat{t}'$ and

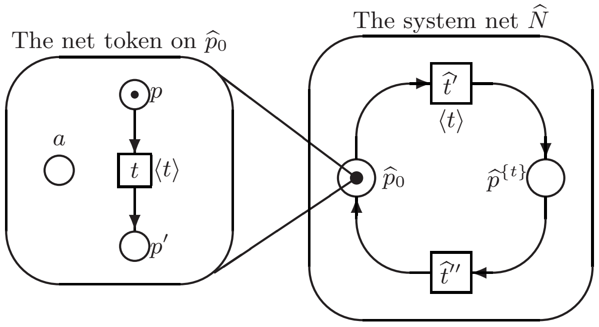


Fig. 5. The simulating net $Inh(N, A)$

$\widehat{t}''$ and one place $\widehat{p}^{\{t\}}$ in the system net with the arcs $\widehat{p}_0 \to \widehat{t}' \to \widehat{p}^{\{t\}} \to \widehat{t}'' \to \widehat{p}_0$. Θ synchronises each system net transition $\widehat{t}'$ with the transition t in the object net N. The conversion $\leftrightarrow$ is defined to allow the transfer of tokens between p and all (p, Z). The construction is illustrated in Figure 5 for the inhibitor net depicted in Figure 4 (a).

Definition 9. *Given the inhibitor net* (N, A) *define the mobile* Eos

$$Inh(N, A) = ((\widehat{N}, \mathcal{N}, d, \Theta, \mu_0), \leftrightarrow)$$

1. *The system net* $\widehat{N}$ *is given by* $\widehat{P} = \{\widehat{p}_0\} \cup \{\widehat{p}^{\{t\}} \mid t \in T\}$, $\widehat{T} = \{\widehat{t}', \widehat{t}'' \mid t \in T\}$, *and* $\mathbf{pre}(\widehat{t}')(\widehat{p}_0) = \mathbf{pre}(\widehat{t}'')(\widehat{p}^{\{t\}}) = \mathbf{post}(\widehat{t}')(\widehat{p}^{\{t\}}) = \mathbf{post}(\widehat{t}'')(\widehat{p}_0) = 1$ *for all* $t \in T$ *(and* 0 *everywhere else).*
2. *The set of object nets is* $\mathcal{N} = \{N\} \cup \{N_Z \mid Z \in \mathcal{Z}\}$.
3. *The typing is defined* $d(\widehat{p}_0) = N$ *and* $d(\widehat{p}^Z) = N_Z$ *for all* $Z \in \mathcal{Z}$.
4. $\Theta = \left\{(\widehat{t}', C) \mid t \in T \wedge C = \{t\} \cup \{\epsilon_{N_Z} \mid \emptyset \neq Z \in \mathcal{Z}\}\right\} \cup \{(\widehat{t}'', \epsilon_{\mathcal{N}}) \mid t \in T\}$.
5. *The initial marking is* $\mu_0 = (\widehat{p}_0, M_0)$.
6. *The conversion* $\leftrightarrow$ *is defined by the family* $\mathcal{C} = \{C_p \mid p \in P\}$ *of conversion sets* $C_p := \{p\} \cup \{(p, Z) \mid Z \in \mathcal{Z}\}$.

Mobile Eos simulate inhibitor nets directly by $Inh(N, A)$, which also proves Theorem 5.

Theorem 6. *Let* (N, A) *be an inhibitor net. Then we have:*

$$M \xrightarrow[N]{t} M' \iff (\widehat{p}_0, M) \xrightarrow[Inh(N,A)]{(\widehat{t}',C)} \left(\widehat{p}^{\{t\}}, M' \times \{\{t\}\}\right) \xrightarrow[Inh(N,A)]{(\widehat{t}'',\epsilon_{\mathcal{N}})} (\widehat{p}_0, M')$$

Proof. We show that transition t is enabled in marking M for the inhibitor net iff $(\widehat{t}', C)$ where $C = \{t)\} \cup \{\epsilon_{N_Z} \mid \emptyset \neq Z \in \mathcal{Z}\}\}$ is enabled in the marking $(\widehat{p}_0, M)$ for $Inh(N, A)$.

Assume that t is enabled in marking M in (N, A). Consider the marking $(\widehat{p}_0, M)$. In the system net the preset of $\widehat{t}'$ and in the net token the preset of t is sufficiently marked. By the definition of the conversion $\leftrightarrow$ it is possible to convert the marking M of the object net N into a marking of the object net $N_{\{t\}}$ iff $M(p) = 0$ for all tested places $p \in (\_At)$. The marking of the net token is modified from M to $M' \times \{Z\}$ due to the internal action of the net token which corresponds to the firing of t in the inhibitor net,

Conversely, if $(\widehat{t}', C)$ is enabled in the marking $(\widehat{p}_0, M)$ then $M(p) \geq W(p,t)$ for all $p \in {}^{\bullet}t$. By the conversion we obtain $M(p) = 0$ for all $(p) \in (\_At)$ since there is no corresponding place (p, Z) in N_Z. Hence, t is enabled in the inhibitor net for marking M. Since the system net of $Inh(N, A)$ is a state machine when firing $(\widehat{t}', C)$ the place $\widehat{p}^Z$ is marked, enabling only $\widehat{t}''$. It is clear that $(\widehat{t}', C)(\widehat{t}'', \epsilon_{\mathcal{N}})$ is the only firing sequence.

The correct correspondence of the markings $M' \times \{\{t\}\}$ and M' follows from the fact that the conversion relation modifies only the second component of the marking while the first reflects the marking of the inhibitor net. Since the initial marking is $\mu_0 = (\widehat{p}_0, M_0)$ we have a simulation of the inhibitor net.

5 Name Spaces and Mobile EOS

Given a mobility system (OS, LS) we define a mobile EOS $Mob(OS)$ that allows global access only within a location. The main idea is similar to the construction of $\mathrm{R}\text{N}(OS)$. As it is shown by Theorem 1, the global access which is defined by reference semantics is characterised by the P/T net $\mathrm{R}\text{N}(OS)$ which is obtained by fusing the system with all the object nets according to the synchronisation relation. The resulting net describes one single name spaces.

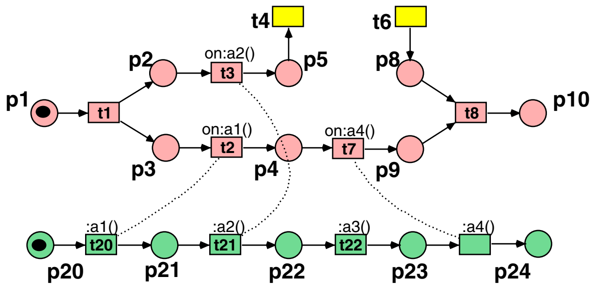


Fig. 6. The Mobile Agent in Building A

If we have different name spaces defined by a locality infrastructure, we fuse only the subnet of the system net that describes a name space: Each location $L \in \mathcal{L}$ of the locality infrastructure is fused with the object nets. If we fuse the name space named *Building A* of Figure 1 with the agent's object net then we obtain the P/T net of Figure 6. (Note, that we kept the graphical layout and indicated the transition fusion by dotted lines).

Given a locality infrastructure $LS = (\mathcal{L}, N_m)$ the new system net $\widehat{N}'$ has one place p_L for each location $L \in \mathcal{L}$: $\widehat{P}' = \{p_L \mid L \in \mathcal{L}\}$. The transitions are the mobility transitions: $\widehat{T}' = T_m$. By definition there is exactly one location L for each $t_m \in T_m$ such ${}^\bullet t_m \subseteq P_L$. Analogously, there is exactly one L' such that $t_m{}^\bullet \subseteq P_{L'}$. For all mobility transitions $t_m \in T_m$ that connect L with L' we add p_L and $p_{L'}$ as side conditions of t_m in the new system net.

For each location net $L \in \mathcal{L}$ we define the object net $N_{\mathcal{N},L}$ that is the union of L with the object nets $N \in \mathcal{N}$ (where each node is tagged with L to make the nets $N_{\mathcal{N},L}$ disjoint). Each system net place p_L carries tokens of type $N(\mathcal{N}, L)$. The net $N(\mathcal{N}, L)$ models agents within the name space L.

The movement transitions $t_m \in T_m$ in the system net $\widehat{N}'$ are synchronised by Θ with the corresponding transition (t_m, L) of the object net $N_{\mathcal{N},L}$.

The initial marking μ_0 can be denoted in the form $\mu_0 = \sum_{L\in\mathcal{L}} \mu_L$ where $\mu_L = \sum_{i=1}^{|\mu_L|}(p_{L,i}, M_{L,i})$ and $p_{L,i} \in P_L$ for all L and i. The new Eos marks each place p_L with marking $\sum_{i=1}^{|\mu_L|} p_{L,i} + (M_{L,i} \times \{L\})$, which is the corresponds exactly with the marking μ_L in the object net $N_{\mathcal{N},L}$.

The conversion $\leftrightarrow$ is defined by the conversion sets $E_{N,p}$ for all $N \in \mathcal{N}$. Each conversion set $E_{N,p} := \{(p, L) \mid L \in \mathcal{L}\}$ contains the new object net places (p, L) that describe the same place p – only at different locations L.

Definition 10. *Let (OS, LS) be a mobility system with $OS = (\widehat{N}, \mathcal{N}, d, \Theta, \mu_0)$ and $LS = (\mathcal{L}, N_m)$. Define the mobile* EOS

$$Mob(OS) := ((\widehat{N}', \mathcal{N}', d', \Theta', \mu_0'), \leftrightarrow)$$

where:

1. *The system net is $\widehat{N}' = (\widehat{P}', \widehat{T}', \widehat{\mathbf{pre}}', \widehat{\mathbf{post}}')$ where $\widehat{P}' = \{p_L \mid L \in \mathcal{L}\}$, $\widehat{T}' = T_m$, and $\mathbf{pre}(t_m)(p_L) = \mathbf{post}(t_m)(p_L) = 1$ if ${}^\bullet t_m \subseteq P_L \vee t_m{}^\bullet \subseteq P_L$ and 0 otherwise.*
2. *The set of object nets is $\mathcal{N}' = \{N_{\mathcal{N},L} \mid L \in \mathcal{L}\}$ where $N_{\mathcal{N},L} = (P, T, \mathbf{pre}, \mathbf{post})$ is defined for each location $L = (P_L, T_L, \mathbf{pre}_L, \mathbf{post}_L) \in \mathcal{L}$ by:*

$$\begin{aligned} P &:= P_L \cup \textstyle\bigcup_{N\in\mathcal{N}}(P_N \times \{L\}) \\ T &:= \{(\widehat{\tau}, C, L) \mid (\widehat{\tau}, C) \in \Theta \wedge \widehat{\tau} \in T_L\} \\ &\quad\ \cup \{(t_m, L) \mid t_m \in T_m \wedge (t_m{}^\bullet \subseteq P_L \vee {}^\bullet t_m \subseteq P_L)\} \end{aligned}$$

$$\begin{aligned} \mathbf{pre}(\widehat{\tau}, C, L)) &= \mathbf{pre}(\widehat{\tau}) + \textstyle\sum_{\tau\in C} \mathbf{pre}(\tau) \\ \mathbf{pre}((t_m, L))(p) &= \begin{cases} \mathbf{pre}_L(t_m)(p), & \text{if } p \in P_L \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

   Analogously for **post***.*
3. *Each system net place p_L carries tokens of type $N_{\mathcal{N},L}$, i.e. $d(p_L) = N_{\mathcal{N},L}$.*
4. *$\Theta = \big\{(t_m, C) \mid t_m \in T_m \wedge C = \{(t_m, L) \mid \ (t_m{}^\bullet \subseteq P_L \vee {}^\bullet t_m \subseteq P_L)\}\big\} \cup \{\epsilon_{N_{\mathcal{N},L}} \mid N_{\mathcal{N},L} \in \mathcal{N}' \wedge \neg(t_m{}^\bullet \subseteq P_L \vee {}^\bullet t_m \subseteq P_L)\}$*
5. *The initial marking is $\mu_0' = \sum_{L\in\mathcal{L}}(p_L), \sum_{i=1}^{|\mu_L|} p_{L,i} + (M_{L,i} \times \{L\}))$, where $\mu_0 = \sum_{L\in\mathcal{L}} \mu_L$ and $\mu_L = \sum_{i=1}^{|\mu_L|}(p_{L,i}, M_{L,i})$, such that $p_{L,i} \in P_L$ for all L and $1 \le i \le |\mu_L|$.*
6. *The conversion $\leftrightarrow$ is defined by the family of conversion sets $\mathcal{E} = \{E_{A,p} \mid A \in \mathcal{A}, p \in P_A\}$ with $E_{A,p} := \{(p, L) \mid L \in \mathcal{L}\}$.*

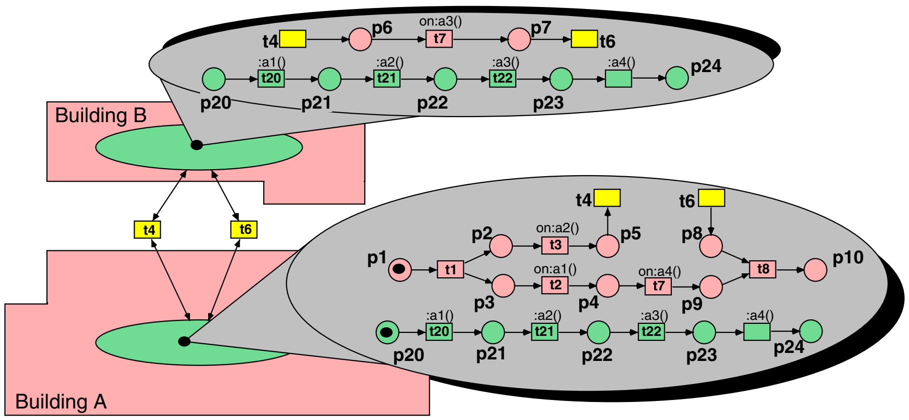


Fig. 7. A mobile agent's environment

The resulting mobile Eos constructed from the Eos in Figure 1 is shown in Figure 7. Each building is represented by a separate place. These two places are connected by the mobility transitions t_4 and t_6. The mobility transitions t_4 and t_6 are also present in each agent net allowing the transfer between buildings. The subnets defined by P_A and T_A as well as P_B and T_B describe the buildings. They are now present as part of the agent net. The agent's and the buildings' events are synchronised via channel inscriptions.

The mobile Eos $Mob(OS)$ defines the desired behaviour: Within the same location L, an object net has global access to all the resources of the name space, since they are all in the same net: $N_{\mathcal{N},L}$. Different locations are isolated since they are in different net tokens.

6 Conclusion

In this paper we have investigated mobile agents that act in a distributed name space. There is a fundamental difference between belonging to a name space and migrating between name spaces. An object belonging to a name space can be accessed directly via pointers, but when migrating between name spaces, objects have to be treated as values that can be copied into network messages.

For the modelling of mobile systems it is essential that the formalism used supports both representations. To accomplish this, we have defined mobile Eos, a generalisation of the well-established formalism of Eos. The main extension is the use of conversion equivalences for the firing rule.

We have shown that mobile Eos subsume reference and value semantics. Furthermore we have proved that they are strictly more expressive than Eos: While boundedness is decidable for Eos it is not for mobile Eos since it is possible to construct an mobile Eos which simulates a given inhibitor net. We have also showed in an example that mobile Eos are suitable for expressing the intended behaviour for distributed name spaces.

References

[BBPP04] Bednarczyk, M.A., Bernardinello, L., Pawlowski, W., Pomello, L.: Modelling mobility with Petri hypernets. In: Fiadeiro, J.L., Mosses, P.D., Orejas, F. (eds.) WADT 2004. LNCS, vol. 3423, pp. 28–44. Springer, Heidelberg (2004)

[Bus99] Busi, N.: Mobile nets. Formal Methods for Open Object-Based Distributed Systems, pp. 51–66 (1999)

[CGG00] Cardelli, L., Gordon, A.D., Ghelli, G.: Ambient groups and mobility types. Technical report, Microsoft Research and University of Pisa (2000)

[Hir02] Hiraishi, K.: PN^2: An elementary model for design and analysis of multi-agent systems. In: Arbab, F., Talcott, C.L. (eds.) COORDINATION 2002. LNCS, vol. 2315, pp. 220–235. Springer, Heidelberg (2002)

[KF06] Köhler, M., Farwer, B.: Modelling global and local name spaces for mobile agents using object nets. Fundamenta Informaticae 72(1-3), 109–122 (2006)

[KMR01] Köhler, M., Moldt, D., Rölke, H.: Modeling the behaviour of Petri net agents. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 224–241. Springer, Heidelberg (2001)

[KMR03] Köhler, M., Moldt, D., Rölke, H.: Modelling mobility and mobile agents using nets within nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 121–140. Springer, Heidelberg (2003)

[Köh06] Köhler, M.: The reachability problem for object nets. In: Moldt, D. (ed.) Proceedings of the Workshop on Modelling, object, components, and agents (MOCA'06). University of Hamburg, Department for Computer Science (2006)

[KR03] Köhler, M., Rölke, H.: Concurrency for mobile object-net systems. Fundamenta Informaticae, vol. 54(2-3) (2003)

[KR04] Köhler, M., Rölke, H.: Properties of Object Petri Nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 278–297. Springer, Heidelberg (2004)

[KWD$^+$04] Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: Renew. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 484–493. Springer, Heidelberg (2004)

[Lak05] Lakos, C.: A Petri net view of mobility. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 174–188. Springer, Heidelberg (2005)

[Lom00] Lomazova, I.A.: Nested Petri nets – a formalism for specification of multi-agent distributed systems. Fundamenta Informaticae 43(1-4), 195–214 (2000)

[MPW92] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts 1-2. Information and computation 100(1), 1–77 (1992)

[Val91] Valk, R.: Modelling concurrency by task/flow EN systems. In: 3rd Workshop on Concurrency and Compositionality, number 191 in GMD-Studien, St. Augustin, Bonn, Gesellschaft für Mathematik und Datenverarbeitung (1991)

[Val98] Valk, R.: Petri nets as token objects: An introduction to elementary object nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–25. Springer, Heidelberg (1998)

[Val03] Valk, R.: Object Petri nets: Using the nets-within-nets paradigm. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Advanced Course on Petri Nets 2003. LNCS, vol. 3098, pp. 819–848. Springer, Heidelberg (2003)

[VC98] Vitek, J., Castagna, G.: Seal: A framework for secure mobile computations. In: ICCL Workshop: Internet Programming Languages, pp. 47–77 (1998)

[XD00] Xu, D., Deng, Y.: Modeling mobile agent systems with high level Petri nets. In: IEEE International Conference on Systems, Man, and Cybernetics 2000 (2000)

Web Service Orchestration with Super-Dual Object Nets

Michael Köhler and Heiko Rölke

University of Hamburg, Department for Informatics
Vogt-Kölln-Str. 30, 22527 Hamburg
{koehler,roelke}@informatik.uni-hamburg.de

Abstract. Refinement of Petri nets is well suited for the hierarchical design of system models. It is used to represent a model at different levels of abstraction.

Usually, refinement is a static concept. For some inherent dynamic domains as for example the modelling of web services, we need a more flexible form of refinement, e.g. to bind web services at run-time. Run-time binding allows for a flexible orchestration of services. The requirement of dynamic refinement at run-time is quite strong. Since we would like to redefine the system structure by itself, transition refinement cannot be implemented by a model transformation. Instead, an approach is needed which allows for dynamic net structures that can evolve as an effect of transitions firing.

In previous work we introduced nets-within-nets as a formalism for the dynamic refinement of tokens. Here we consider an extension of nets-within-nets that uses special net tokens describing the refinement structure of transitions. Using this formalism it is possible to update refinements, introduce alternative refinements, etc. We present some formal properties of the extended formalism and introduce an example implementation for the tool Renew.

Keywords: duality, refinement, nets-within-nets, Petri nets, super-dual nets.

1 Motivation: Web Service Orchestration

Web services [ACKM03] are a standard means to integrate services distributed over the Internet using standard web technologies like HTTP, XML, SOAP, OWL, WSDL, BPEL4WS etc. (cf. [Got00, OAS07]). Conceptually, this approach is very similar to remote procedure calls in CORBA [COR07]. The main difference is that web services focus on the semantic level of services using semantic web techniques (formal ontologies for data and processes). The semantic level allows for an automated dynamic binding depending on the available services. This enables the programmer to concentrate on the *orchestration* of web services.

To give an example, have a look a the Petri net in Figure 1 which defines a simple web service workflow: The scenario describes the organisation of a journey from S to D. This task is split into two major sub-tasks that are executed in

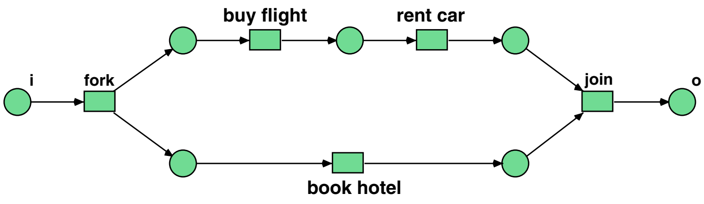


Fig. 1. Web Service Workflow

parallel. The first sub-task organises a flight from S to an airport A that is close to the final destination D and a car to travel from the airport A to D. The second task is to book a hotel at D.

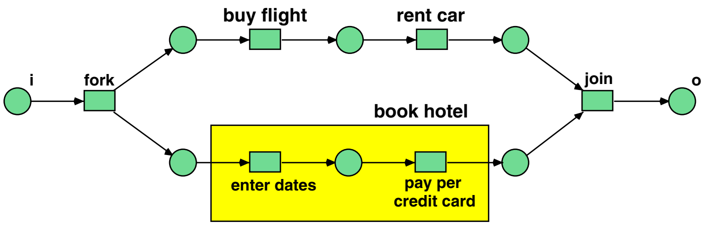


Fig. 2. Refinement of the Web Service

For each transition of the net in Figure 1 we use a specific web service. We assume that each web service itself is modelled by a Petri net (cf. [KMO06], [Mar05], [NM03]). Then the resulting system model is obtained by refinement. This approach corresponds to the usual top-down design of a software product. Figure 2 shows the refinement of the transition *book hotel* where the travel dates have to be filled in before the hotel is booked.

A Petri net formalism supporting transition refinement has the advantage over other formalisms that the original abstract net of the early design stages does not have to be redefined, but is continuously used for the later models and for the implementation models. An example of such a formalism is the one of (hierarchical) Coloured Petri Nets [Jen92] used in the Design/CPN tool [Des07].

Unfortunately the proposed transition refinement procedures in [Jen92] only support static refinements that cannot be changed at runtime. For web services the refinement is however not static. Usually we have a look-up services which provides a whole repository of services that can be used alternatively. Usually the user can decide which service is best for him with respect to costs, timing, convenience etc. Figure 3 shows a repository with three subnets that can be used for the refinement of the transition *book hotel*. The rhombic nodes named *(un)bind web service* are used to formalise this dynamic refinement. The intended meaning is the following: whenever the action *bind web service* is executed one subnet is removed from the *web service repository* and the subnet is used as

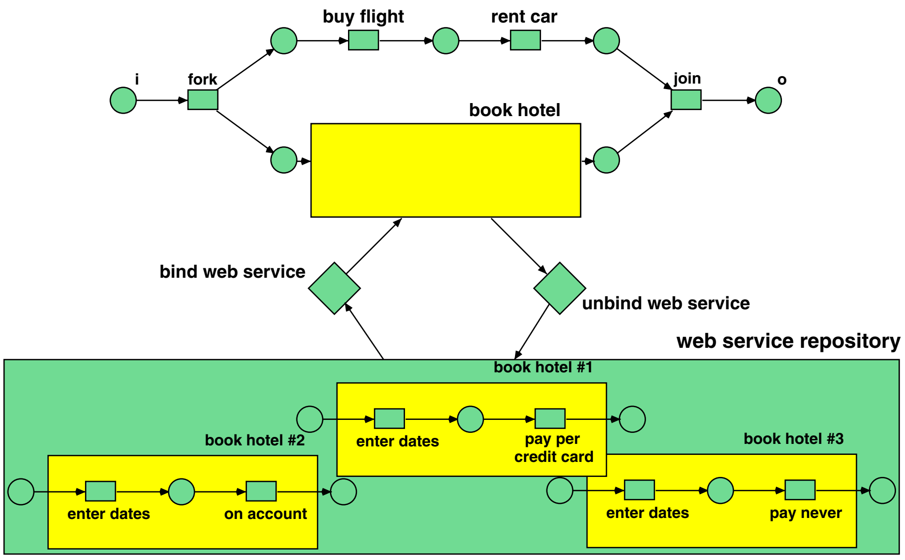


Fig. 3. Dynamic Refinement using a Repository

the refinement of the transition *book hotel*. The action *unbind web service* is the reverse operation.

In the formalism developed in this article, the rhombic nodes are used like places since they connect transitions. We regard the rhombic nodes as places to avoid introducing additional modellings constructs.
The "only" novelties our new formalism require, are *marked transitions* and the *firing of places*(sic!).[1] Also we use *nets as tokens*. The nets with these properties are called *Super-Dual Object Nets*. They are a variant of the *nets-within-nets* approach of Valk [Val03], therefor the term *Object Nets*; they are called *super-dual* because places are also marked and are able to fire.

The remaining sections are structured as follows: In Section 2 we introduce our approach of marked transitions for Petri nets. We define super-dual nets and their firing rule. In Section 3 we describe how the concept of super-dual nets can be lifted to object nets and give an definition of the new formalism of super-dual object nets. In Section 4 we describe how super-dual object nets can be simulated by object nets. Section 5 explains a first attempt to integrate dynamic transition refinement in Renew [Kum01, KWD$^+$04]. The paper ends with a conclusion.

2 Introduction to Super-Dual Petri Nets

This section starts with a short remainder of Petri nets basics. This is to avoid notational confusions. After that, super-dual nets will be introduced.

[1] Note, that the repository is a proper transition with the implementing nets as its marking.

2.1 Basic Definitions

The definition of Petri nets relies on the notion of multi-sets. A multi-set on the set D is a mapping $\mathbf{A} : D \to \mathbb{N}$. The set of all mapings from D to $\mathbb{N}$ is denoted by $\mathbb{N}^D$. Multi-sets are generalisations of sets in the sense that every subset of D corresponds to a multi-set $\mathbf{A}$ with $\mathbf{A}(x) \leq 1$ for all $x \in D$. The empty multi-set $\mathbf{0}$ is defined as $\mathbf{0}(x) = 0$ for all $x \in D$. The carrier of a multi-set $\mathbf{A}$ is $\mathrm{dom}(\mathbf{A}) := \{x \in D \mid \mathbf{A}(x) > 0\}$. The cardinality of a multi-set is $|\mathbf{A}| := \sum_{x \in D} \mathbf{A}(x)$. A multi-set $\mathbf{A}$ is called *finite* iff $|\mathbf{A}| < \infty$. The multi-set sum $\mathbf{A} + \mathbf{B}$ is defined as $(\mathbf{A} + \mathbf{B})(x) := \mathbf{A}(x) + \mathbf{B}(x)$, the difference $\mathbf{A} - \mathbf{B}$ by $(\mathbf{A} - \mathbf{B})(x) := \max(\mathbf{A}(x) - \mathbf{B}(x), 0)$. Equality $\mathbf{A} = \mathbf{B}$ is defined element-wise: $\forall x \in D : \mathbf{A}(x) = \mathbf{B}(x)$. Multi-sets are partially ordered: $\mathbf{A} \leq \mathbf{B} \iff \forall x \in D : \mathbf{A}(x) \leq \mathbf{B}(x)$. The strict order $\mathbf{A} < \mathbf{B}$ holds iff $\mathbf{A} \leq \mathbf{B}$ and $\mathbf{A} \neq \mathbf{B}$. The notation is overloaded, being used for sets as well as multi-sets. The meaning will be apparent from its use.

In the following we assume all multi-sets to be finite. A finite multi-set $\mathbf{A}$ can be considered as the formal sum $\mathbf{A} = \sum_{x \in D} \mathbf{A}(x) \cdot x = \sum_{i=1}^{n} x_i$. Finite multi-sets are the freely generated commutative monoid. If the set D is finite, then a multi-set $\mathbf{A} \in \mathbb{N}^D$ can be represented equivalently as a vector $\mathbf{A} \in \mathbb{N}^{|D|}$.

Any mapping $f : D \to D'$ can be generalised to a homomorphism $f^\sharp : \mathbb{N}^D \to \mathbb{N}^{D'}$ on multi-sets: $f^\sharp \left(\sum_{i=1}^{n} a_i\right) = \sum_{i=1}^{n} f(a_i)$. This includes the special case $f^\sharp(\mathbf{0}) = \mathbf{0}$. These definitions are in accordance with the set-theoretic notation $f(A) = \{f(a) \mid a \in A\}$. In this paper we simply use f instead of $f^\sharp$.

2.2 Petri Nets

A *Petri net* is a tuple $N = (P, T, F)$ where P is a set of places, T is a set of transitions, disjoint from P, i.e. $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation. Some commonly used notations for Petri nets are ${}^\bullet y := (\_ F\, y)$ for the *preset* and $y^\bullet := (y\, F\, \_)$ for the *postset* of a net element y.

To simplify the definition of duality and conjugation we only consider ordinary Petri nets, i.e. we do not deal with arc weights. The mappings F^-, F^+ are defined by $F^-(t)(p) := |F \cap \{(p,t)\}|$ and $F^+(t)(p) := |F \cap \{(t,p)\}|$.

A marking of a net N is a multi-set of places: $\mathbf{m} \in \mathbb{N}^P$. Places are depicted as circles, transitions as rectangles, and the flow relation as arcs between the nodes. The marking is visualised as $\mathbf{m}(p)$ tokens on the place p.

A *marked Petri net* is a tuple $(N, \mathbf{m}_0)$ consisting of a Petri net and a start marking. Throughout this paper, we speak of *Petri nets* or simply *nets* instead of (ordinary) *marked Petri nets.*

A multi-set of transitions $\mathbf{u} \in \mathbb{N}^T$ of a net N is *enabled* in marking $\mathbf{m}$ iff $\forall p \in P : \mathbf{m}(p) \geq F^-(\mathbf{u})(p)$ holds. The enablement of $\mathbf{u}$ in marking $\mathbf{m}$ is denoted by $\mathbf{m} \xrightarrow{\mathbf{u}}$. A transition multi-set $\mathbf{u}$ enabled in $\mathbf{m}$ can fire in the successor marking $\mathbf{m}'$ where $\mathbf{m}'(p) = \mathbf{m}(p) - F^-(u)(p) + F^+(u)(p)$. Firing is denoted by $\mathbf{m} \xrightarrow{\mathbf{u}} \mathbf{m}'$.

Using multi-set operators $\mathbf{m} \xrightarrow{\mathbf{u}}$ is equivalent to $\mathbf{m} \geq F^-(\mathbf{u})$, and the successor marking is $\mathbf{m}' = \mathbf{m} - F^-(\mathbf{u}) + F^+(\mathbf{u})$.

2.3 Super-Dual Nets

Super-dual nets have been introduced in [KR06]. A super-dual net contains a G-flow (short: a glow) $G \subseteq (P \times T \cup T \times P)$ as an additional structure. G connects places and transitions the same way as the flow F, but with a different semantics (see below).

Definition 1. *A* super-dual net *is a tuple* $SD = (P, T, F, G)$ *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 the flow relation, and*
- $G \subseteq (P \times T \cup T \times P)$ *is the glow relation.*

The preset w.r.t. the glow G is ${}^{\blacksquare}y := (\_ G\, y)$ and the postset is $y^{\blacksquare} := (y\, G \_)$. Analogously to the flow mappings we define the glow mappings $G^-, G^+ : T \to (P \to \mathbb{N})$ by $G^-(t)(p) := |G \cap \{(p,t)\}|$ and $G^+(t)(p) := |G \cap \{(t,p)\}|$.

In super-dual nets, also the transitions may be marked. A marking of a super-dual net is a multi-set of places and transitions: $\mathbf{m} \in \mathbb{N}^{(P \cup T)}$. The tokens on transitions are called *pokens*. A poken is visualised as a little filled square. A marked super-dual net is denoted as $(P, T, F, G, \mathbf{m})$.

For super-dual nets, the firing rule considers the firing of transitions as well as the firing of places.

1. A marking $\mathbf{m}$ enables a transition t only if its preset ${}^{\bullet}t$ is marked and t itself is marked. For a transition multi-set $\mathbf{u} \in \mathbb{N}^T$ we define enabling by:
$$\begin{array}{ll} \mathbf{m}(p) \geq F^-(\mathbf{u})(p) & \text{for all } p \in P \\ \mathbf{m}(t) \geq \mathbf{u}(t) & \text{for all } t \in T \end{array}$$
This means, that the number of pokens $\mathbf{m}(t)$ limits the maximal concurrency of the transition t. Thus $\mathbf{m}(t) = 0$ describes a disabled transition.
2. Conversely, a marking $\mathbf{m}$ enables a place p only if its preset ${}^{\blacksquare}p$ is marked and p itself is marked. For a place multi-set $\mathbf{u} \in \mathbb{N}^P$ we define enablement by:
$$\begin{array}{ll} \mathbf{m}(p) \geq \mathbf{u}(p) & \text{for all } p \in P \\ \mathbf{m}(t) \geq G^-(\mathbf{u})(t) & \text{for all } t \in T \end{array}$$

Example 1. Cf. the net in Figure 4. The place p_5 is connected by glow arcs (the dashed ones) with transitions t_1 and t_2. In the depicted marking, only the transition t_1 is enabled – more exactly: it is enabled twice. Despite the fact, that the preset of transition t_2 is marked, it is not enabled, since t_2 itself is unmarked. Firing of p_5 transfers a poken from t_1 to t_2, and t_2 is then enabled.

Both cases – firing of transitions and of places – can occur in a single step.

Definition 2. *A multi-set of places and transitions* $\mathbf{u} \in \mathbb{N}^{(P \cup T)}$ *of a super-dual net SD is* enabled *in the marking* $\mathbf{m} \in \mathbb{N}^{(P \cup T)}$*, denoted by* $\mathbf{m} \xrightarrow{\mathbf{u}}$*, iff*
$$\begin{array}{ll} \mathbf{m}(p) \geq F^-(\mathbf{u}|_T)(p) + \mathbf{u}(p) & \text{for all } p \in P \text{ and} \\ \mathbf{m}(t) \geq G^-(\mathbf{u}|_P)(t) + \mathbf{u}(t) & \text{for all } t \in T. \end{array}$$

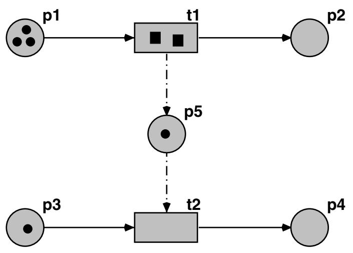


Fig. 4. The super-dual net SD

An enabled multi-set $\mathbf{u}$ *can fire, denoted by* $\mathbf{m} \xrightarrow{\mathbf{u}} \mathbf{m}'$, *resulting in the successor marking* $\mathbf{m}'$ *defined by*

$$\begin{aligned}\mathbf{m}'(p) &= \mathbf{m}(p) - F^-(\mathbf{u}|_T)(p) + F^+(\mathbf{u}|_T)(p)\\ \mathbf{m}'(t) &= \mathbf{m}(t) - G^-(\mathbf{u}|_P)(t) + G^+(\mathbf{u}|_P)(t).\end{aligned}$$

Define $\mathbf{pre}(\mathbf{u}) := F^-(\mathbf{u}|_T) + G^-(\mathbf{u}|_P)$ and $\mathbf{post}(\mathbf{u}) := F^+(\mathbf{u}|_T) + G^+(\mathbf{u}|_P)$. Using multi-set notations $\mathbf{m} \xrightarrow{\mathbf{u}}$ is equivalent to $\mathbf{m} \geq \mathbf{pre}(\mathbf{u}) + \mathbf{u}$. The successor marking is $\mathbf{m}' = \mathbf{m} - \mathbf{pre}(\mathbf{u}) + \mathbf{post}(\mathbf{u})$.

Duality. Given a super-dual net $SD = (P, T, F, G, \mathbf{m})$, the dual net (interchanging transitions and places) is defined as $SD^d := (T, P, F, G, \mathbf{m})$ and the conjugated net (interchanging flow and glow) is $SD^c := (P, T, G, F, \mathbf{m})$. Note, that also the dual of a marking can be considered for super-dual nets. We have the commutativity: $SD^{cd} = SD^{dc}$. For the super-dual net of Figure 4 these constructions are illustrated in Figure 5.

The following property justifies the name “super-dual nets”.

Proposition 1. *Let SD be a super-dual net. SD corresponds to* SD^{cd}*:*

$$\mathbf{m} \xrightarrow[SD]{\mathbf{u}} \mathbf{m}' \iff \mathbf{m} \xrightarrow[SD^{cd}]{\mathbf{u}} \mathbf{m}'$$

Proof. Simultanously interchanging P and T as well as F and G in Definition 2 is the identity transformation.

Components. We define the F-component $SD|_F$ and the G-component $SD|_G$ of a marked super-dual net $SD = (P, T, F, G, \mathbf{m})$ as:

$$SD|_F := (P, T, F, \mathbf{m}|_P) \tag{1}$$
$$SD|_G := (P, T, G, \mathbf{m}|_T) \tag{2}$$

Both constructions are illustrated in Figure 6. Note, that the components $SD|_F$ and the dual of the G-component, $SD|_G^d$ (but not $SD|_G$ itself) are Petri nets.

The following proposition relates the behaviour of a super-dual net to that of its components.

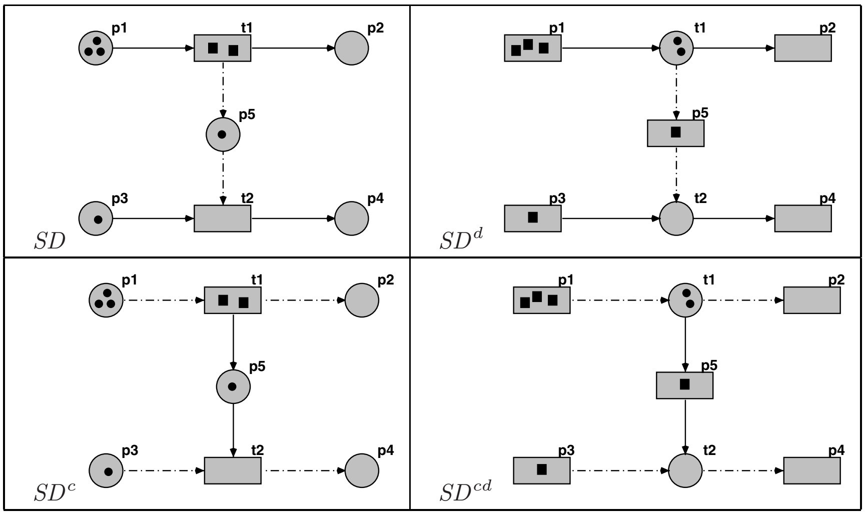


Fig. 5. Duality and Conjugation

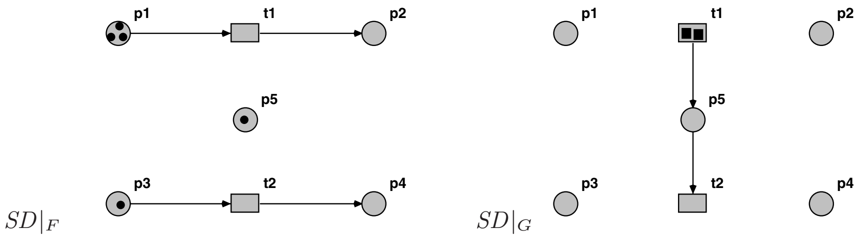


Fig. 6. The F- and the G-component

Proposition 2. *Let* $\mathbf{m} \in \mathbb{N}^{(P \cup T)}$ *be a marking of a super-dual net* $SD = (P, T, F, G)$. *Let* $\mathbf{u} \in \mathbb{N}^{(P \cup T)}$. *Then for the* F*-component and the dual of the* G*-component we have (For the proof see [KR06]):*

$$\forall \mathbf{u} \in \mathbb{N}^T : \mathbf{m} \xrightarrow[SD]{\mathbf{u}} \mathbf{m}' \iff \left(\mathbf{m}|_P \xrightarrow[SD|_F]{\mathbf{u}} \mathbf{m}'|_P \wedge \mathbf{m}|_T = \mathbf{m}'|_T \geq \mathbf{u} \right)$$
$$\forall \mathbf{u} \in \mathbb{N}^P : \mathbf{m} \xrightarrow[SD]{\mathbf{u}} \mathbf{m}' \iff \left(\mathbf{m}|_T \xrightarrow[SD|_G^d]{\mathbf{u}} \mathbf{m}'|_T \wedge \mathbf{m}|_P = \mathbf{m}'|_P \geq \mathbf{u} \right)$$

3 From Object Nets to Super-Dual Object Nets

We are interested in a dynamic refinement of transitions, i.e. a refinement that can be changed at runtime. This change should be made by the net itself. Our basic approach is to regard sub-nets as special tokens of transitions. As mentioned in the introduction this approach proposes two extensions to the Petri net formalism: (1) Petri nets can be used as tokens and (2) transitions may be marked.

The first extension of Petri nets to object nets – also known as the *nets-within-nets* approach – has been proposed by Valk [Val91, Val03], and further developed e.g. in [Far99], [Lom00], and [KR03, KR04]. The Petri nets that are used as tokens are called *net-tokens.* Net-tokens are tokens with internal structure and inner activity. This is different from place refinement, since tokens are transported while a place refinement is static. Net-tokens are some kind of *dynamic* refinement of states.

3.1 Object Nets and Object Net Systems

In the following we give a condensed definition of object net systems. For simplicity reasons we abstract from the syntax of inscriptions of net elements and synchronisations as it is used for reference nets [Kum01] in the Renew tool (cf. [KWD+04]).

Object net systems have the set of object nets as their colour set. In [KR03] we generate the net-tokens via instantiation from a finite set of nets. In this definition, we assume for simplicity reasons an arbitrary set of object nets:

$$\mathcal{N} = \{N_0, N_1, \ldots\}$$

One object net models black tokens: $\bullet \in \mathcal{N}$. This net has one initally unmarked place and no transitions.

In coloured nets, each transition t fires according to a mode b generated from transition guards, arc expressions and variable assignments. Let B be the set of firing modes. Each object net is a tuple

$$N = (P_N, T_N, F_N^-, F_N^+)$$

where $F_N^-, F_N^+ : T_N \to (B \to (P_N \to \mathbb{N}^{\mathcal{N}}))$. Given a binding b $F_N^-(t)(b)(p)$ is a multiset of object nets.

Let P denote the union of all place components: $P := \bigcup_{N\in\mathcal{N}} P_N$. Assume analogously defined union sets for transitions T, etc.

A marking μ of an object net system maps each place to a multi-set of object nets:

$$\mu : P \to \mathbb{N}^{\mathcal{N}}$$

Here $\mu(p)(N) > 0$ describes the fact, that the place p is marked with $\mu(p)(N)$ net-tokens of the type N.

Transitions in different nets may be synchronised via channels. In Renew, channels are also used to exchange parameters. Each transition has at most one uplink, which is passive, and several downlinks, which are active in the sense that they choose the synchronisation partner. Due to this structure we obtain tree-like synchronisations. The formal definition is based on the synchronisation trees. The set of all synchronisation trees is $\mathcal{T} = \bigcup_{n\geq 0} \mathcal{T}_n$ where

$$\mathcal{T}_n := \{(t,b)[\theta_1 \cdots \theta_k] \mid t \in T \wedge b \in B \wedge \forall 1 \leq i \leq k : \theta_i \in \textstyle\bigcup_{l<n} \mathcal{T}_l\}. \qquad (3)$$

The predomain F^- (and analogously for F^+) is extended to $\widehat{F}^- : \Theta \to (P_N \to \mathbb{N}^{\mathcal{N}})$ by:

$$\widehat{F}^-((t,b)[\theta_1 \cdots \theta_k]) = F^-(t)(b) + \sum\nolimits_{i=1}^{k} \widehat{F}^-(\theta_i) \qquad (4)$$

Definition 3. *An* Object Net System *is a tuple* $OS = (\mathcal{N}, \Theta, \mu_0)$ *where*

– $\mathcal{N}$ *is a set of object nets,*
– $\Theta \subseteq \mathcal{T}$ *is the set of system events, and*
– μ_0 *is the initial marking.*

As usual we have $\mu \xrightarrow{\theta} \mu'$ iff $\mu \geq \widehat{F}^-(\theta)$ and $\mu' = \mu - \widehat{F}^-(\theta) + \widehat{F}^+(\theta)$. This firing rule describes the reference semantics of object nets – for an in-deep comparison of alternative firing rules cf. [Val03, KR04].

3.2 Super-Dual Object Nets

Similarly to the extension of Petri nets to super-dual nets in section 2, we extend the object net formalism by using nets as pokens, called *net-pokens*. The net-pokens can be used as a dynamic *refinement* (similar to a sub routine) of the transitions they mark. Figure 7 shows a super-dual object net with nets on places and on transitions.

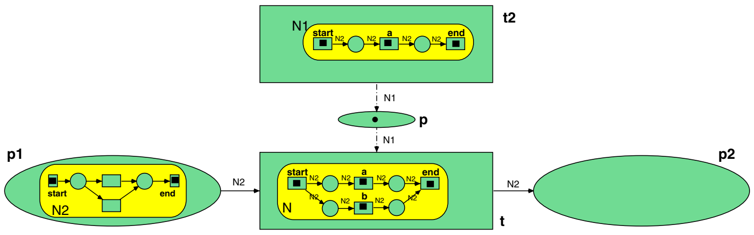


Fig. 7. A Petri net with nets as tokens for places and transitions

Since these refinements are defined as markings it is possible to move net-pokens using the token-game of object nets. In Figure 7 the place p “fires” the net-poken from t_2 to t. Transition t is then marked by two net-pokens, which means that there are two modes of refinement for t. The equivalent net containing the conflict between the possible refinement is given in Figure 8.

Each object net is a super-dual object net $N = (P_N, T_N, F_N^-, F_N^+, G_N^-, G_N^+)$ where $G_N^{\pm} : T_N \to (B \to (P_N \to \mathbb{N}^{\mathcal{N}}))$ define the inscriptions for glow arcs. One net models black pokens: $\blacksquare \in \mathcal{N}$. Each net-poken $N \in \mathcal{N}$ has one transition start_N with empty preset and one transition end_N with empty postset. These transitions are used to start (or end, respectively) the dynamic refinement implemented by the net-poken. From a practical point of view it is reasonable to require that the net is unmarked when the refinement is started (i.e. when transition start_N fires) and is unmarked again when it is ended by end_N. We do not adopt such a restriction here to allow a general definition.

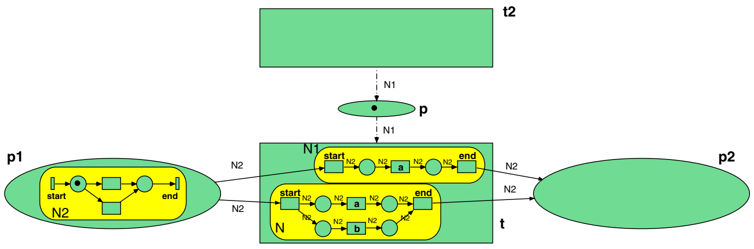


Fig. 8. Equivalent refinement after firing of place p

A marking μ of a super-dual object net system maps each place and transition to a multi-set of object nets:

$$\mu : (P \cup T) \to \mathbb{N}^{\mathcal{N}}$$

In analogy to $\mathcal{T}$ we define a tree structure of places: $\mathcal{P} = \bigcup_{n\geq 0} \mathcal{P}_n$ where

$$\mathcal{P}_n = \{(p,b)[\pi_1 \cdots \pi_k] \mid p \in P \wedge b \in B \wedge \forall 1 \leq i \leq k : \pi_i \in \bigcup_{l<n} \mathcal{P}_l\}. \quad (5)$$

The mappings $\widehat{G}^-$ and $\widehat{G}^+$ are defined analogously to $\widehat{F}^-$ and $\widehat{F}^+$.

The mapping $\nu : (\mathcal{P} \cup \mathcal{T}) \to \mathbb{N}^{(P\cup T)}$ constructs a multiset by removing the nesting structure:

$$\nu(x[\xi_1 \cdots \xi_k]) := x + \sum\nolimits_{i=1}^{k} \nu(\xi_i) \quad (6)$$

Here, $\nu(\theta)(t,b)$ is the number of occurences of (t,b) in the nested structure θ.

A dynamically refined transition t is enabled in a mode N where N is the net-poken implementing the refinement. If N is the black poken $\blacksquare$, then this transition is not refined and synchronisation is possible. If N is not the black poken, it is used as a dynamic refinement of t. This refinement splits t into a *start* and an *end* part: $(t, b, N, \mathit{start})$ and (t, b, N, end).

$$\mathcal{R} = \{(t, b, N, \alpha) \mid t \in T \wedge b \in B \wedge \blacksquare \neq N \in \mathcal{N}, \alpha \in \{\mathit{start}, \mathit{end}\}\} \quad (7)$$

Definition 4. *A Super-Dual Object Net System* $\mathit{SDOS} = (\mathcal{N}, \Theta, \mu_0)$ *consists of the following components:*

- $\mathcal{N}$ *is a set of object nets,*
- $\Theta \subseteq (\mathcal{T} \cup \mathcal{P} \cup \mathcal{R})$ *is the set of system events, and*
- $\mu_0 : (P \cup T) \to \mathbb{N}^{\mathcal{N}}$ *is the initial marking.*

The set of system events Θ contains elements from $\mathcal{T}$, $\mathcal{P}$, and $\mathcal{R}$. So, we have different kinds of firing modes:

1. $\theta \in \Theta \cap \mathcal{T}$: As usual we have $\mu \xrightarrow{\theta} \mu'$ iff $\forall p \in P : \mu(p) \geq \widehat{F}^-(\theta)(p)$ and $\mu(t)(\blacksquare) \geq \sum_{b\in B} \nu(\theta)(t,b)$. Then $\mu' = \mu - \widehat{F}^-(\theta) + \widehat{F}^+(\theta)$.
2. $\theta \in \Theta \cap \mathcal{P}$: As usual we have $\mu \xrightarrow{\pi} \mu'$ iff $\forall t \in T : \mu(t) \geq \widehat{G}^-(\pi)(t)$ and $\mu(p)(\bullet) \geq \sum_{b\in B} \nu(\theta)(p,b)$. Then $\mu' = \mu - \widehat{G}^-(\pi) + \widehat{G}^+(\pi)$.

3. $\theta \in \Theta \cap \mathcal{R}$: A dynamic refinement has two parts.
   - The control is carried over from t to a refining net-poken N:

$$\mu \xrightarrow{(t,b,N,start)} \mu' \iff \begin{array}{l} \forall p \in P : \mu(p) \geq F^-(t)(b)(p) \\ \wedge\ \mu(t)(N) \geq 1 \wedge \mu(\mathsf{start}_N)(\blacksquare) \geq 1 \\ \wedge\ \mu' = \mu - F^-(t)(b) + F^+(\mathsf{start}_N)(b) \end{array}$$

   - The control is given back from N to t:

$$\mu \xrightarrow{(t,b,N,end)} \mu' \iff \begin{array}{l} \forall p \in P : \mu(p) \geq F^-(\mathsf{end}_N)(b)(p) \\ \wedge\ \mu(t)(N) \geq 1 \wedge \mu(\mathsf{end}_N)(\blacksquare) \geq 1 \\ \wedge\ \mu' = \mu - F^-(\mathsf{end}_N)(b) + F^+(t)(b) \end{array}$$

Proposition 3. *Object nets are a special case of super-dual object nets: Each object net system is simulated by a super-dual object net system.*

Proof. The super-dual object net system is obtained from the object net system adding no glow arcs marking all transitions with enough black-pokens. To allow all synchronisations $\theta \in \Theta$, the transition t is marked with $\max\{\nu(\theta)(t,b) \mid b \in B, \theta \in \Theta\}$ black-pokens. Then the super-dual object net system behaves the same way as the object net system since we have no refinements and no events $\theta \in \Theta \cap \mathcal{P}$. All the events $\theta \in \Theta \cap \mathcal{T}$ are enabled correspondingly and the effect is the same as for the object net system.

4 Simulating Super-Dual Object Nets

In our previous work [KR06] we have shown that super-dual nets can simulate Petri nets and, more interesting, that Petri nets can simulate super-dual nets – both in respect to the possible firing sequences. The construction uses the dual of the G-component (i.e. $SD|_G^d$), renames all nodes x to $x^{(d)}$ and combines it with the F-component. The result is the simulating Petri net $N(SD)$ (Figure 9 illustrates the construction for the net SD of Figure 4).

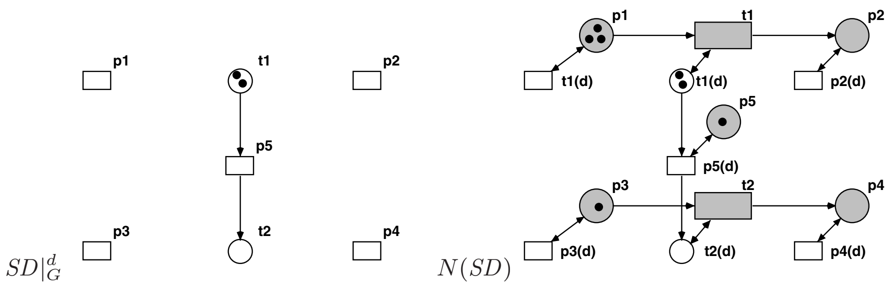


Fig. 9. The simulating net $N(SD)$

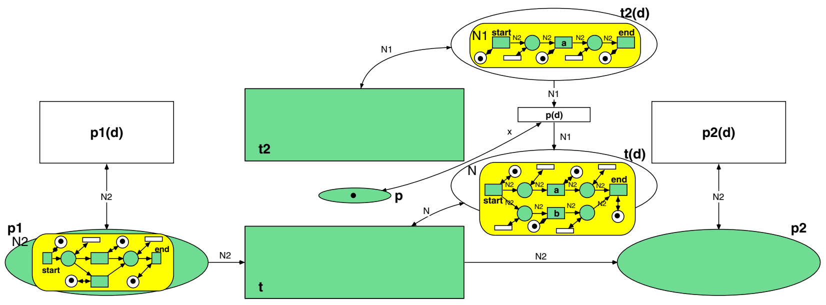


Fig. 10. Adding the dual component to the net of Figure 7

We will now lift our results to object nets and super-dual object nets to come back to the goal of this work, dynamic refinement of transitions. The construction for super-dual object nets is similar to the construction of $N(SD)$ for super-dual nets. We illustrate the construction of the simulating object net $OS(SDOS)$ at the example net from Figure 7. The construction involves two steps: In the first step the dual of the G-component (the unfilled nodes) is added to the F-component (the filled nodes) as side conditions for each object net. The refining net-pokens become net-tokens. The resulting net is given in Figure 10. Note, that the side transitions named $p_1^{(d)}$ and $p_2^{(d)}$ have no effect. The same holds for the side condition named p. These nodes might be omitted. In the second step we split each transition t of the F-component into two parts: t_{start} and t_{end}, similarly to the construction suggested in the introduction. Transition t_{start} synchronises with the input transition start_N, i.e. it starts the refining subnet N. Similarly, t_{end} synchronises with the output transition. The resulting net – omitting synchronisation inscriptions – is given in Figure 11.

We formalise this dualisation construction in the following. The element in the dual component corresponding to $n \in P \cup T$ is denoted $n^{(d)}$. The mapping f^P maps each transition t to its dual, i.e. the place $t^{(d)}$ and each place p to its

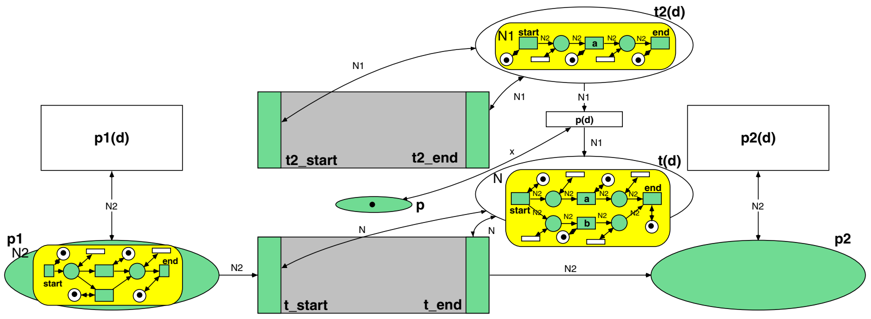


Fig. 11. Adding the start/end structure

dual, i.e. the place $p^{(d)}$: $f^X(n) = n$ if $n \in X$ and $f^X(n) = n^{(d)}$ if $n \notin X$. The notation extends to pairs: $f^X((a,b)) = (f^X(a), f^X(b))$ and to sets: $f^X(A) = \{f^X(a) \mid a \in A\}$. This definition also extends to the nested structures $\mathcal{P}$ and $\mathcal{T}$ the usual way: $f^X(x[\xi_1 \dots \xi_k]) := f^X(x)[f^X(\xi_1) \dots f^X(\xi_k)]$.

For each marking μ in the super-dual object net the marking in the simulating net $\widetilde{\mu}$ is defined by $\widetilde{\mu}(p) = \mu(p)$ and $\widetilde{\mu}(t) = \mu(t^{(d)})$ or shorter for $n \in P \cup T$:

$$\widetilde{\mu}(n) = \mu(f^P(n)) \tag{8}$$

The simulating event $\widetilde{\theta}$ is defined according to the three kinds of firing: a synchronisation of transitions $\theta \in \Theta \cap \mathcal{T}$ is simulated by θ. A synchronisation of places $\theta \in \Theta \cap \mathcal{P}$ is simulated by $\theta^{(d)}$. Both cases are subsumed by the definition $\widetilde{\theta} := f^T(\theta)$. The start event $(t, b, N, start) \in \Theta \cap \mathcal{R}$ is simulated by the synchronisation of t_{start} (i.e. the first part of t) with the starting transition start_N of the refining net N:

$$(t_{start}, b)[(\mathsf{start}_N, b)[]]$$

Similarly for the event (t, b, N, end). This leads to the following definition:

$$\widetilde{\theta} := \begin{cases} f^T(\theta), & \text{if } \theta \in \Theta \cap (\mathcal{T} \cup \mathcal{P}) \\ (t_\alpha, b)[(\alpha_N, b)[]], & \text{if } \theta = (t, b, N, \alpha) \in \Theta \cap \mathcal{R}, \alpha \in \{start, end\} \end{cases} \tag{9}$$

The notation extends to sets: $\widetilde{\Theta} = \{\widetilde{\theta} \mid \theta \in \Theta\}$.

Definition 5. *Given a super-dual object system $SDOS = (\mathcal{N}, \Theta, \mu_0)$ we define the object net system*

$$OS(SDOS) = (\{\widetilde{N} \mid N \in \mathcal{N}\}, \widetilde{\Theta}, \widetilde{\mu}_0)$$

where $\widetilde{N} = (f^P(P \cup T), f^T(P \cup T), \widetilde{F^-}, \widetilde{F^+})$ and with the bindings $\widetilde{B} = B \times \mathcal{N}$ the pre- and post conditions are defined by:

$$\widetilde{F^\pm}(t)(b,N)(n) = \begin{cases} F^\pm(t)(b)(p), & \textit{if } n = p \in P \\ N, & \textit{if } n = t^{(d)}, t \in T^{(d)} \\ \mathbf{0}, & \textit{otherwise} \end{cases}$$

and

$$\widetilde{F^\pm}(p^{(d)})(b,N)(n) = \begin{cases} G^\pm(p)(b)(t), & \textit{if } n = t^{(d)} \in T^{(d)} \\ N, & \textit{if } n = p \in P \\ \mathbf{0}, & \textit{otherwise} \end{cases}$$

Then we have the following simulation property:

Proposition 4. *Let SDOS be a super-dual object net. For the object net system $OS(SDOS)$ we have:*

$$\mu \xrightarrow[SDOS]{\theta} \mu' \iff \widetilde{\mu} \xrightarrow[OS(SDOS)]{\widetilde{\theta}} \widetilde{\mu}'$$

Proof. It is easy to observe from the construction, that whenever an event θ of $SDOS$ is enabled in μ then the event $\widetilde{\theta}$ of is enabled in $\widetilde{\mu}$ in the simulation object net system $OS(SDOS)$. This holds for the three cases of firing modes. No other events are enabled and the successor markings correspond.

This approach of expressing dynamic transition refinement using object nets is (partially) implemented as a special construct in our tool Renew. In the following we will illustrate the tool extension for the domain of dynamic workflows.

5 Transition Refinement: A First Approach in Renew

The concept of dynamic refinement is especially valuable in the context of workflow management systems. Dynamic refinements can be used to replace a sub workflow with a more up-to-date version respecting some preservation rules, like workflow inheritance (cf. [vdAB02]).

A first attempt to implement dynamic transition refinement in the Petri net tool Renew was done in the so-called workflow plug-in[2] [JKMUN02]. Among various means for the definition and execution of workflows a so-called *task transition* was implemented. The task transition does not exactly meet our design criteria for dynamic refinement, but comes close enough to take a look.

The task transition implements two features. The first is irrelevant for the topic of this paper: the execution of a task transition may be canceled (see [JKMUN02]). The second feature is close to the desired behaviour of a refined transition postulated in Section 1.

Statically, a task transition looks like a normal transition with bold lines at the left and right side of its rectangle figure – see for example transition book hotel in the net travelWorkflow in the middle of Figure 12. It is inscribed with a triple consisting of a task (bookHotel), i.e. the net[3] that refines the transition, a set of parameters (hotelData) to pass to this net and the expected result that should be passed back. It is important to notice, that the task associated to a task transition need not be statically associated but may be exchanged at runtime. The task transition therefore puts dynamic transition refinement down to dynamic place refinement in terms of nets-within-nets.

When firing a task transition, the transition gets marked with the subnet that refines it. Renew treats this refinement token just as an ordinary token, so that the usual means for inspection and manipulation are available.

What is missing to fully meet our design criteria is – besides some implementation details – a better support for the separation of net refinement tokens from other tokens. This could be done in terms of a net type hierarchy.

[2] Renew offers a powerful plug-in concept making it easy to implement new functionality in all areas of net design, simulation and analysis.

[3] Note, that in the workflow implementation a task is not necessarily a net, but may also be Java code.

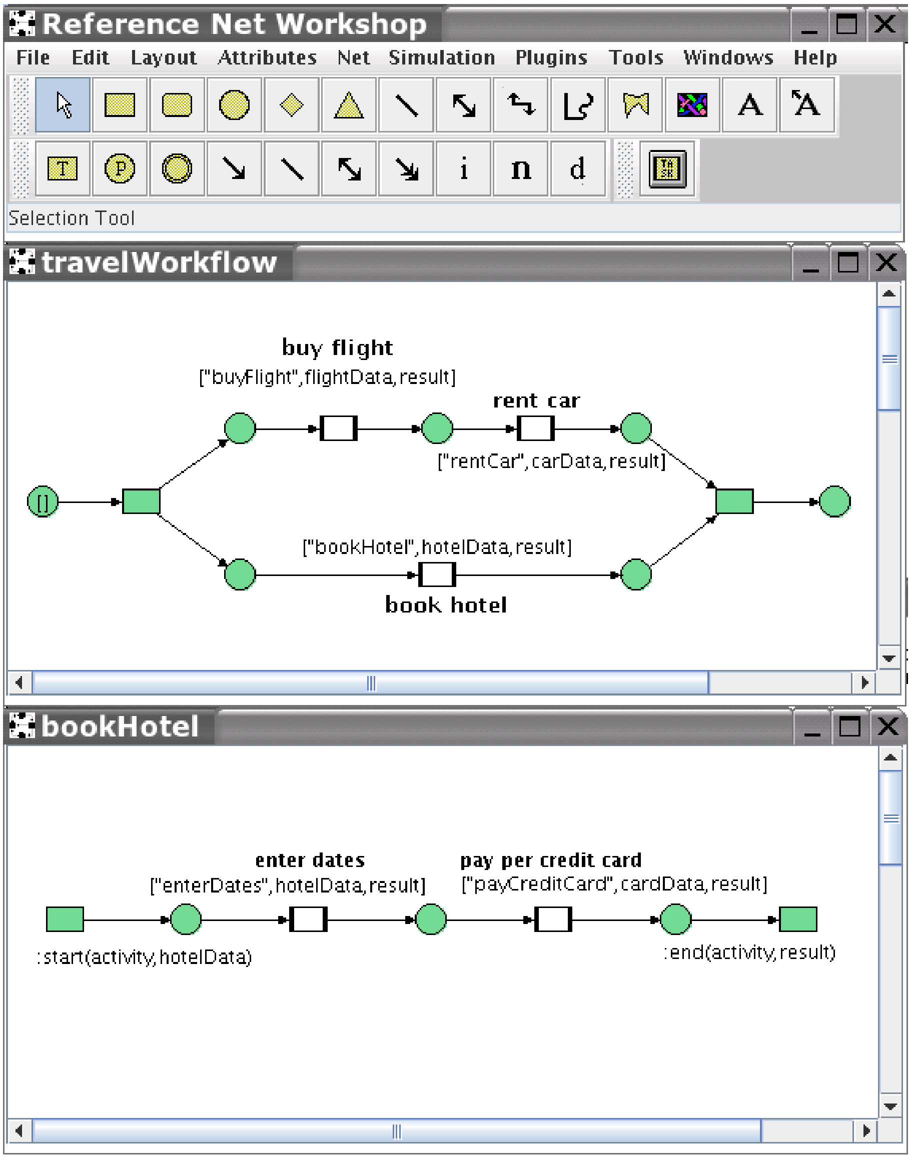

Reference Net Workshop
File Edit Layout Attributes Net Simulation Plugins Tools Windows Help
Selection Tool
travelWorkflow
buy flight
["buyFlight",flightData,result]
rent car
["rentCar",carData,result]
[]
["bookHotel",hotelData,result]
book hotel
bookHotel
enter dates
["enterDates",hotelData,result]
pay per credit card
["payCreditCard",cardData,result]
:start(activity,hotelData)
:end(activity,result)

Fig. 12. The workflow modelled with task transitions

6 Related Work

To the best of our knowledge there are no publications describing dynamic transition refinements for Petri nets. There exists, however, a small amount of publications on Petri nets that can modify their structure at runtime and, separated from these, on duality in Petri nets.

Our approach describes a special kind of Petri nets that can modify their structure via dynamic refinement of transitions. A first approach to the ability of structure changing at runtime are self-modifying nets [Val78] which allow for arc weights that are marking depending. A special case is the empty marking that temporarily deletes arcs from the net. Another approach to structure modification in Petri nets is that of mobile nets [Bus99], algebraic nets with graph rewriting [HEM05], and recursive nets [HP99].

Duality is an important concept in Petri's general net theory and is discussed in [GLT80]. Petri only considers unmarked nets, so no "problems" with tokens on transitions arise. The restriction to unmarked nets is renounced by Lautenbach [Lau03]. However, his concept of duality differs from the one presented in this paper. He considers firing in the dual reverse net N^{rd}. In his approach transitions become marked and places fire the tokens, which are lying on transitions, in the reversed arc direction. Additionally, contrary to our approach, for his definition a token on a transition disables its firing.

7 Conclusion

In this presentation we studied the dynamic refinement of transitions. Following the ideas of extending Petri nets to super-dual nets, we generalised object net systems to super-dual object systems. Super-dual object net systems are nets-within-nets allowing nets as tokens both on places and on transitions. Transition marking nets, called net-pokens, may be moved around from one transition to another. They refine the transition they are actually marking. This offers the desired properties of a dynamic, run-time refinement procedure that is controlled by the net itself.

Super-dual object nets are related to an implementation of a workflow extension plugin of the Renew tool. This extension has a special notion of dynamically refinable transitions, called tasks. These task transitions are executed by instantiating a net-token that implements a sub-workflow. In accordance with our definitions these sub-workflows are the dual of normal workflows, i.e. they have a unique input transition and a unique output transition. The workflow management system can make use of this mechanism when replacing sub-workflows by updates at runtime. This can be done easily by moving net-tokens around or creating new ones at runtime, e.g. as a result of a planning process. One can think of mobile workflows implemented by mobile agents in the style of [KMR03].

References

[ACKM03] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. Springer, Heidelberg (2003)

[Bus99] Busi, N.: Mobile nets. Formal Methods for Open Object-Based Distributed Systems, pp. 51–66 (1999)

[COR07] Common object request broker architecture (1993-2007) www.omg.org/corba

[Des07] The Design CPN homepage http://www.daimi.au.dk/designCPN (2007)

[Far99] Farwer, B.: A linear logic view of object Petri nets. Fundamenta Informaticae 37(3), 225–246 (1999)

[GLT80] Genrich, H.J., Lautenbach, K., Thiagarajan, P.S.: Elements of general net theory. In: Brauer, W. (ed.) Net Theory and Applications. LNCS, vol. 84, pp. 21–163. Springer, Heidelberg (1980)

[Got00] Gottschalk, K.: Web services architecture overview. Whitepaper, IBM developerWorks (2000)

[HEM05] Hoffmann, K., Ehrig, H., Mossakowski, T.: High-level nets with nets and rules as tokens. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 268–288. Springer, Heidelberg (2005)

[HP99] Haddad, S., Poitrenaud, D.: Theoretical aspects of recursive Petri nets. In: Donatelli, S., Kleijn, J. (eds.) Application and Theory of Petri Nets 1999. LNCS, vol. 1630, pp. 228–247. Springer, Heidelberg (1999)

[Jen92] Jensen, K.: Coloured Petri nets, Basic Methods, Analysis Methods and Practical Use. In: EATCS monographs on theoretical computer science, Springer, Heidelberg (1992)

[JKMUN02] Jacob, T., Kummer, O., Moldt, D., Ultes-Nitsche, U.: Implementation of workflow systems using reference nets – security and operability aspects. In: Jensen, K. (ed.) Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools. University of Aarhus, Department of Computer Science (2002)

[KMO06] Köhler, M., Moldt, D., Ortmann, J.: Dynamic service composition: A petri-net based approach. In: Manolopoulos, Y., Filipe, J., Constantopoulos, P., Cordeiro, J. (ed.) Conference on Enterprise Information Systems: Databases and Information Systems Integration (ICEIS 2006), pp. 159–165 (2006)

[KMR03] Köhler, M., Moldt, D., Rölke, H.: Modelling mobility and mobile agents using nets within nets. In: van der Aalst, W., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 121–140. Springer, Heidelberg (2003)

[KR03] Köhler, M., Rölke, H.: Concurrency for mobile object-net systems. Fundamenta Informaticae, vol. 54(2-3) (2003)

[KR04] Köhler, M., Rölke, H.: Properties of Object Petri Nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 278–297. Springer, Heidelberg (2004)

[KR06] Köhler, M., Rölke, H.: Properties of super-dual nets. Fundamenta Informaticae 72(1-3), 245–254 (2006)

[Kum01] Kummer, O.: Introduction to Petri nets and reference nets. Sozionikaktuell, vol. 1 (2001)

[KWD$^+$04] Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: Renew. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 484–493. Springer, Heidelberg (2004)

[Lau03] Lautenbach, K.: Duality of marked place/transition nets. Technical Report 18, Universität Koblenz-Landau (2003)

[Lom00] Lomazova, I.A.: Nested Petri nets – a formalism for specification of multiagent distributed systems. Fundamenta Informaticae 43(1-4), 195–214 (2000)

[Mar05] Martens, A.: Analyzing web service based business processes. In: Cerioli, M. (ed.) FASE 2005. Held as Part of the Joint Conferences on Theory and Practice of Software, ETAPS 2005, LNCS, vol. 3442, pp. 19–33. Springer, Heidelberg (2005)

[NM03] Narayanan, S., McIlraith, S.: Analysis and simulation of web services. Computer Networks 42(5), 675–693 (2003)

[OAS07] Organization for the advancement of structured information standards (1993–2007) www.oasis-open.org

[Val78] Valk, R.: Self-modifying nets, a natural extension of Petri nets. In: Ausiello, G., Böhm, C. (eds.) Automata, Languages and Programming. LNCS, vol. 62, pp. 464–476. Springer, Heidelberg (1978)

[Val91] Valk, R.: Modelling concurrency by task/flow EN systems. In: 3rd Workshop on Concurrency and Compositionality, number 191 in GMD-Studien, St. Augustin, Bonn, Gesellschaft für Mathematik und Datenverarbeitung (1991)

[Val03] Valk, R.: Object Petri nets: Using the nets-within-nets paradigm. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Advanced Course on Petri Nets 2003. LNCS, vol. 3098, pp. 819–848. Springer, Heidelberg (2003)

[vdAB02] van der Aalst, W., Basten, T.: Inheritance of workflows: An approach to tackling problems related to change. Theoretical Computer Science 270(1-2), 125–203 (2002)

Synthesis of Elementary Net Systems with Context Arcs and Localities

Maciej Koutny and Marta Pietkiewicz-Koutny

School of Computing Science, Newcastle University
Newcastle upon Tyne, NE1 7RU, United Kingdom
{maciej.koutny,marta.koutny}@newcastle.ac.uk

Abstract. We investigate the synthesis problem for ENCL-systems, defined as Elementary Net Systems extended with context (inhibitor and activator) arcs and explicit event localities. Since co-located events are meant to be executed synchronously, the behaviour of such systems is captured by step transition systems, where arcs are labelled by sets of events rather than by single events. We completely characterise transition systems generated by ENCL-systems after extending the standard notion of a region — defined as a certain set of states — with explicit information about events which, in particular, are responsible for crossing its border. As a result, we are able to construct, for each such transition system, a suitable ENCL-system generating it.

Keywords: theory of concurrency, Petri nets, elementary net systems, localities, net synthesis, step sequence semantics, structure and behaviour of nets, theory of regions, transition systems, inhibitor arcs, activator arcs, context arcs.

1 Introduction

We are concerned with a class of concurrent computational systems whose dynamic behaviours exhibit a particular mix of *asynchronous* and *synchronous* executions, and are often described as adhering to the 'globally asynchronous locally synchronous' (or GALS) paradigm. Intuitively, actions which are 'close' to each other are executed synchronously and as many as possible actions are always selected for execution. In all other cases, actions are executed asynchronously. Two important applications of the GALS approach can be found in hardware design, where a VLSI chip may contain multiple clocks responsible for synchronising different subsets of gates [1], and in biologically motivated computing, where a membrane system models a cell with compartments, inside which reactions are carried out in co-ordinated pulses [2]. In both cases, the activities in different localities can proceed independently, subject to communication and/or synchronisation constraints.

To formally model GALS systems, [3] introduced *Place/Transition-nets with localities* (PTL-nets), defined as PT-nets where transitions are assigned to explicit localities. Each locality identifies transitions which may only be executed synchronously and in a maximally concurrent manner. The idea of adding localities to an existing Petri net model was taken further in [4], where Elementary Net Systems (EN-systems) replaced PT-nets as an underlying system model. In this paper, we build on the work reported in [4], by considering EN-systems extended with two non-standard kinds of arcs, namely *inhibitor* arcs and *activator* (or *read*) arcs, collectively referred to as *context* arcs following the terminology of [5]. The resulting model will be referred to as *Elementary Net Systems with Context Arcs and Localities* (or ENCL-systems).

It is worth pointing out that both inhibitor arcs (capturing the idea the enabling of a transition depends on a place being *unmarked*) and activator arcs (capturing the idea the enabling of a transition depends on a place being *marked* by more tokens than those consumed when the transition is fired) are presumably the most prominent extensions of the basic Petri net model considered in the literature. Such context arcs can be used to test for a specific condition, rather than producing and consuming resources, and proved to be useful in areas such as communication protocols [6], performance analysis [7], and concurrent programming [8]. More recently, [9] applied context arcs to deal with several salient behavioural features of membrane systems, such as promoters, inhibitors and dissolving as well as thickening of membranes.

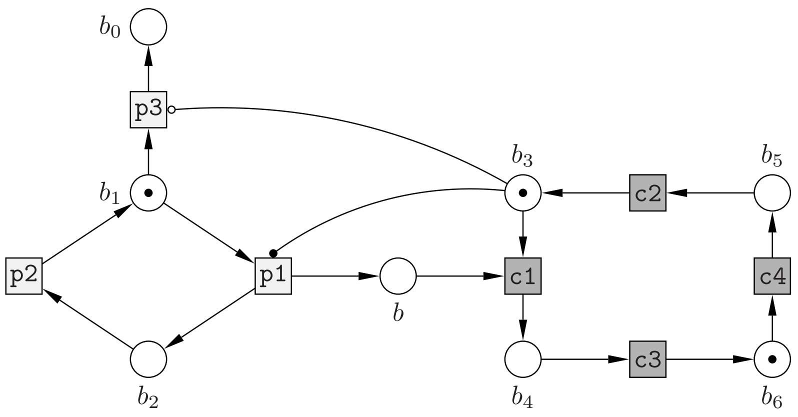


Fig. 1. A producer/consumer system with a business conscious producer

Consider the ENCL-system in Figure 1 modelling a producer/consumer system consisting of one producer (who can execute events `p1`, `p2` and `p3`), and two consumers (who can execute events `c1`, `c2`, `c3` and `c4`). The buffer-like condition b in the middle holds items produced by the event `p1` and consumed by `c1`. The activator arc between `p1` and b_3 (represented by an edge ending with a small black circle) means that the producer adds a new item to the buffer only if there is a consumer waiting for it, and the inhibitor arc between `p3` and b_3 (represented by an edge ending with a small circle) means that the producer can leave the production cycle only when no customer is eager to get the produced items. It is assumed that all the events executed by the producer belong to one locality, while all the events executed by the consumers belong to another locality. To indicate this in the diagram, we use different shading for the boxes representing events assigned to different localities.

In terms of possible behaviours, adding localities can have a significant impact on both the executability of events and reachability of global states. For example, under the standard net semantics the model in Figure 1 would be able to execute the step sequence $\{\mathtt{p1}\}\{\mathtt{c1}\}$, but the execution model of ENCL-systems rejects this. The reason is that after $\{\mathtt{p1}\}$, it is possible to execute the step $\{\mathtt{c1}, \mathtt{c4}\}$ consisting of two co-located events, and so executing c1 alone would violate the maximal concurrency execution rule within the locality assigned to the events used by the consumers. A possible way of 'repairing' this step sequence would be to add the 'missing' event, resulting in the legal step sequence $\{\mathtt{p1}\}\{\mathtt{c1}, \mathtt{c4}\}$. Another legal step sequence is $\{\mathtt{p1}\}\{\mathtt{p2}, \mathtt{c1}, \mathtt{c4}\}$, where the second step is maximally concurrent in a global sense, as it cannot be extended any further. Note also that the event p3 is not enabled at the beginning since b_3 contains a token and there is an inhibitor arc linking p3 and b_3, and after $\{\mathtt{p1}\}\{\mathtt{p2}, \mathtt{c1}, \mathtt{c4}\}$ the event p1 is not enabled since b_3 is now empty and there is an activator arc linking p1 and b_3.

The Petri net problem we are going to investigate in this paper, commonly referred to as the *Synthesis Problem*, is in essence concerned with model transformation, from a class of transition systems (sometimes called reachability graphs) to a class of Petri nets. The key requirement is that the Petri net obtained from a given transition system should capture the same behaviour, i.e., its reachability graph should be isomorphic to the original transition system. This problem was solved for the class of EN-systems in [10], using the notion of a region which links nodes of transition systems (global states) with conditions in the corresponding nets (local states). The solution was later extended to the pure bounded PT-nets [11], general Petri nets [12], safe nets [13] and EN-systems with inhibitor arcs [14,15], by adopting the original definition of a region or using some extended notion of a generalised region [16].

In a previous paper [4], we have solved the synthesis problem for the class of Elementary Net Systems with Localities (ENL-systems). In doing so, we introduced *io-regions*, a generalisation of the standard notion of a region of a transition system, as the latter proved to be insufficient to deal with the class of ENL-systems (and hence also for ENCL-systems considered in this paper). To explain the idea behind io-regions, consider the transition system shown in Figure 2(a), which is isomorphic to the reachability graph of the ENL-system shown in Figure 2(b). (Note that the two events there, e and f, are assumed to be co-located.) The standard region-based synthesis procedure would attempt to construct the conditions of the net in Figure 2(b), by identifying each of these with the set of the nodes of the transition system where it 'holds'. For example, the region corresponding to b_1 comprises just one state, $r = \{s_{init}\}$. Similarly, $r' = \{s\}$ is a region where b_2 holds. (Note that there are two more 'trivial' regions, $\{s_{init}, s\}$ and $\varnothing$, which are ignored by the synthesis procedure.) However,

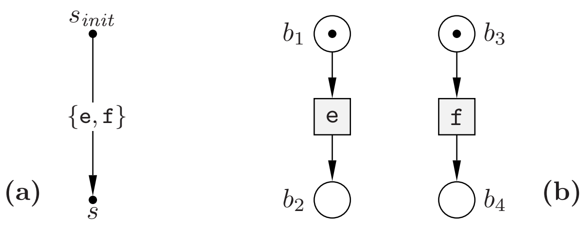


Fig. 2. A transition system with co-located transitions $\mathtt{e}$ and $\mathtt{f}$ **(a)**, and a corresponding ENL-system **(b)**

this is not enough to construct the ENL-system in Figure 2(b) since there are only two non-trivial regions and we need four, one for each of the conditions.

An intuitive reason why the standard construction does not work for the transition system in Figure 2(a) is that the 'set-of-states' notion of region is not rich enough for the purposes of synthesising ENL-systems. The modification to the original notion proposed in [4] is based on having also explicit *input* and *output* events of a set of states, which point at those events which are 'responsible' for entering the region and for leaving it. More precisely, an io-regions is a triple: $\mathfrak{r} = (in, r, out)$, where r is a set of states, in is a set of events which are responsible for entering r, and out is a set of events which are responsible for leaving r. In the case of the example in Figure 2(a), one can find four non-trivial io-regions: $\mathfrak{r}_1 = (\varnothing, \{s_{init}\}, \{\mathtt{e}\})$ $\mathfrak{r}_2 = (\{\mathtt{e}\}, \{s\}, \varnothing\})$, $\mathfrak{r}_3 = (\varnothing, \{s_{init}\}, \{\mathtt{f}\})$ and $\mathfrak{r}_4 = (\{\mathtt{f}\}, \{s\}, \varnothing)$. Now one has enough regions to construct the conditions of the ENL-system in Figure 2(b), namely each $\mathfrak{r}_i$ corresponds to b_i.

In this paper, we will extend the idea of an io-region to also cope with context arcs. Briefly, we will base our synthesis solution on *context regions* (or c-regions), each such region being a tuple (r, in, out, inh, act) where the two additional components, inh and act, carry information about events which are related with r due to the presence of a context arc.

The paper is organised as follows. In the next section, we introduce formally ENCL-systems. After that we define ENCL-transition systems and later show that the reachability graphs of ENCL-systems are indeed ENCL-transition systems. We finally demonstrate how to construct an ENCL-system corresponding to a given ENCL-transition system.

2 ENCL-Systems

Throughout the paper we assume that $\mathcal{E}$ is a fixed non-empty set of *events*. Each event e is assigned a *locality* $\mathfrak{L}(e)$, and it is *co-located* with another event f whenever $\mathfrak{L}(e) = \mathfrak{L}(f)$.

Definition 1 (net with context arcs). *A* net with context arcs *is a tuple* $\mathfrak{net} \overset{\mathrm{df}}{=} (B, E, F, I, A)$ *such that B and $E \subseteq \mathcal{E}$ are finite disjoint sets, $F \subseteq (B \times E) \cup (E \times B)$ and $I, A \subseteq B \times E$.*

The meaning and graphical representation of B (conditions), E (events) and F (flow relation) is as in the standard net theory. An *inhibitor* arc $(b, e) \in I$ means that e can be enabled only if b is not marked (in the diagrams, it is represented by an edge ending with a small circle), and an *activator* arc $(b, e) \in A$ means that e can be enabled only if b is marked (in the diagrams, it is represented by an edge ending with a small black circle). In diagrams, boxes representing events are shaded, with different shading being used for different localities (see Figure 1). We denote, for every $x \in B \cup E$,

$$^{\bullet}x \overset{\text{df}}{=} \{y \mid (y, x) \in F\} \qquad x^{\bullet} \overset{\text{df}}{=} \{y \mid (x, y) \in F\}$$

$$^{\blacklozenge}x \overset{\text{df}}{=} \{y \mid (x, y) \in I \cup I^{-1}\} \qquad ^{\blacktriangleleft}x \overset{\text{df}}{=} \{y \mid (x, y) \in A \cup A^{-1}\}$$

and we call the above sets the *pre-elements*, $^{\bullet}x$, *post-elements*, $x^{\bullet}$, *inh-elements*, $^{\blacklozenge}x$, and *act-elements*, $^{\blacktriangleleft}x$. Moreover, we denote

$$^{\bullet}x^{\bullet} \overset{\text{df}}{=} {}^{\bullet}x \cup x^{\bullet} \qquad ^{\bullet}x^{\blacktriangleleft} \overset{\text{df}}{=} {}^{\bullet}x \cup {}^{\blacktriangleleft}x \qquad ^{\blacklozenge}x^{\bullet} \overset{\text{df}}{=} x^{\bullet} \cup {}^{\blacklozenge}x\ .$$

All these notations extend in the usual way (i.e., through the set union) to sets of conditions and/or events. It is assumed that for every event $e \in E$, $e^{\bullet}$ and $^{\bullet}e$ are non-empty sets, and $^{\bullet}e$, $e^{\bullet}$, $^{\blacklozenge}e$ and $^{\blacktriangleleft}e$ are mutually disjoint sets. For the ENCL-system in Figure 1, we have $^{\bullet}b_3 = \{\mathtt{c2}\}$, $b_1^{\bullet} = \{\mathtt{p1}, \mathtt{p3}\}$ and $^{\blacklozenge}\mathtt{p3} = {}^{\blacktriangleleft}\mathtt{p1} = \{b_3\}$.

Definition 2 (ENCL-system). *An* elementary net system with context arcs and localities *(*ENCL*-system) is a tuple* $\mathfrak{encl} \overset{\text{df}}{=} (B, E, F, I, A, c_{init})$ *such that* $\mathfrak{net}_{\mathfrak{encl}} \overset{\text{df}}{=} (B, E, F, I, A)$ *is the underlying net with context arcs, and* $c_{init} \subseteq B$ *is the* initial case*. In general, any subset of* B *is a* case*.*

The execution semantics of $\mathfrak{encl}$ is based on steps of simultaneously executed events. We first define the set of *valid steps*:

$$\mathbb{U}_{\mathfrak{encl}} \overset{\text{df}}{=} \{u \subseteq E \mid u \neq \varnothing \ \wedge \ \forall e, f \in u : \ e \neq f \Rightarrow {}^{\bullet}e^{\bullet} \cap {}^{\bullet}f^{\bullet} = \varnothing\}\ .$$

For the ENCL-system in Figure 1, we have $\{\mathtt{p1}, \mathtt{c2}, \mathtt{c3}\} \in \mathbb{U}_{\mathfrak{encl}}$, but $\{\mathtt{p1}, \mathtt{c1}, \mathtt{c4}\} \notin \mathbb{U}_{\mathfrak{encl}}$ since $\mathtt{p1}^{\bullet} \cap {}^{\bullet}\mathtt{c1} \neq \varnothing$.

A step $u \in \mathbb{U}_{\mathfrak{encl}}$ is *enabled* at a case $c \subseteq B$ if $^{\bullet}u^{\blacktriangleleft} \subseteq c$ and $^{\blacklozenge}u^{\bullet} \cap c = \varnothing$, and there is no step $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{encl}}$ satisfying $\mathfrak{L}(e) \in \mathfrak{L}(u)$, $^{\bullet}e^{\blacktriangleleft} \subseteq c$ and $^{\blacklozenge}e^{\bullet} \cap c = \varnothing$.

For the ENCL-system in Figure 1, we have that $\{\mathtt{p1}, \mathtt{c4}\}$ is a step enabled at the initial case, but $\{\mathtt{p3}, \mathtt{c4}\}$ is not since b_3 belongs to c_{init} and there is an inhibitor arc between $\mathtt{p3}$ and b_3. We also note that $u = \{\mathtt{p2}, \mathtt{c1}\}$ is not enabled at the case $c = \{b_2, b, b_3, b_6\}$ because it can be extended by an event $e = \mathtt{c4}$ according to the definition of enabledness.

The above definition of enabledness is based on an *a priori* condition: the activator and inhibitor conditions of events occurring in a step obey their respective constraints *before* the step is executed. In an *a posteriori* approach (see [5]), the respective properties must also be true *after* executing the step. Yet another definition for enabling when activator arcs (or rather read arcs) are involved is given in [17].

The transition relation of $\mathfrak{net}_{\mathfrak{encl}}$, denoted by $\rightarrow_{\mathfrak{net}_{\mathfrak{encl}}}$, is then given as the set of all triples $(c, u, c') \in 2^B \times \mathbb{U}_{\mathfrak{encl}} \times 2^B$ such that u is enabled at c and $c' = (c \setminus {}^\bullet u) \cup u^\bullet$.

The *state space* of $\mathfrak{encl}$, denoted by $C_{\mathfrak{encl}}$, is the least subset of 2^B containing c_{init} such that if $c \in C_{\mathfrak{encl}}$ and $(c, u, c') \in \rightarrow_{\mathfrak{net}_{\mathfrak{encl}}}$ then $c' \in C_{\mathfrak{encl}}$. The *transition relation* of $\mathfrak{encl}$, denoted by $\rightarrow_{\mathfrak{encl}}$, is then defined as $\rightarrow_{\mathfrak{net}_{\mathfrak{encl}}}$ restricted to $C_{\mathfrak{encl}} \times \mathbb{U}_{\mathfrak{encl}} \times C_{\mathfrak{encl}}$. We will use $c \xrightarrow{u}_{\mathfrak{encl}} c'$ to denote that $(c, u, c') \in \rightarrow_{\mathfrak{encl}}$. Also, $c \xrightarrow{u}_{\mathfrak{encl}}$ if $(c, u, c') \in \rightarrow_{\mathfrak{encl}}$, for some c'. For the ENCL-system in Figure 1:

$$c_{init} \xrightarrow{\{\mathtt{p1}\}}_{\mathfrak{encl}} \{b_2, b, b_3, b_6\} \xrightarrow{\{\mathtt{p2,c1,c4}\}}_{\mathfrak{encl}} \{b_1, b_4, b_5\} \,.$$

Proposition 1 ([4]). *If* $c \xrightarrow{u}_{\mathfrak{encl}} c'$ *then* $c \setminus c' = {}^\bullet u$ *and* $c' \setminus c = u^\bullet$.

3 Step Transition Systems and Context Regions

In this section, we first recall the notion of a general step transition system which, after further restrictions, will be used to provide a behavioural model for ENCL-systems, and introduce the notion of a context region.

Definition 3 (transition system, [18,19]). *A* step transition system *is a triple* $\mathfrak{ts} \overset{\text{df}}{=} (S, T, s_{init})$ *where:*

TSys1	S *is a non-empty finite set of* states.
TSys2	$T \subseteq S \times (2^{\mathcal{E}} \setminus \{\varnothing\}) \times S$ *is a finite set of* transitions.
TSys3	$s_{init} \in S$ *is the* initial state.

Throughout this section, the step transition system $\mathfrak{ts}$ will be fixed. We will denote by $\mathcal{E}_{\mathfrak{ts}}$ the set of all the events appearing in its transitions, i.e.,

$$\mathcal{E}_{\mathfrak{ts}} \overset{\text{df}}{=} \bigcup_{(s,u,s') \in T} u \,.$$

We will denote $s \xrightarrow{u} s'$ whenever (s, u, s') is a transition in T, and respectively call s the *source* and s' the *target* of this transition. We will also say that the step u is *enabled* at s, and denote this by $s \xrightarrow{u}$.

For every event $e \in \mathcal{E}_{\mathfrak{ts}}$, we will denote by T_e the set of all the transitions labelled by steps containing e, $T_e \overset{\text{df}}{=} \{(s, u, s') \in T \mid e \in u\}$, and by U_e the set of all the steps labelling these transitions, $U_e \overset{\text{df}}{=} \{u \mid (s, u, s') \in T_e\}$.

We now introduce a central notion of this paper which is meant to link the nodes of a transition system (global states) with the conditions in the hypothetical corresponding net (local states).

Definition 4 (context region). *A* context region *(or c-region) is a tuple*

$$\mathfrak{r} \overset{\text{df}}{=} (r, in, out, inh, act) \in 2^S \times 2^{\mathcal{E}_{\mathfrak{ts}}} \times 2^{\mathcal{E}_{\mathfrak{ts}}} \times 2^{\mathcal{E}_{\mathfrak{ts}}} \times 2^{\mathcal{E}_{\mathfrak{ts}}}$$

such that the following are satisfied, for every transition $s \xrightarrow{u} s'$ of $\mathfrak{ts}$:

1. *$s \in r$ and $s' \notin r$ imply $|u \cap in| = 0$ and $|u \cap out| = 1$.*
2. *$s \notin r$ and $s' \in r$ imply $|u \cap in| = 1$ and $|u \cap out| = 0$.*
3. *$u \cap inh \neq \varnothing$ implies $s \notin r$.*
4. *$u \cap act \neq \varnothing$ implies $s \in r$.*
5. *$u \cap out \neq \varnothing$ implies $s \in r$ and $s' \notin r$.*
6. *$u \cap in \neq \varnothing$ implies $s \notin r$ and $s' \in r$.*
7. *$in \cap inh = \varnothing$ and $out \cap act = \varnothing$.*

We denote $\|\mathfrak{r}\| \overset{\text{df}}{=} r$, ${}^\bullet\mathfrak{r} \overset{\text{df}}{=} in$, $\mathfrak{r}^\bullet \overset{\text{df}}{=} out$, ${}^\blacklozenge\mathfrak{r} \overset{\text{df}}{=} inh$ and ${}^\blacktriangleleft\mathfrak{r} \overset{\text{df}}{=} act$.

The step transition system shown in Figure 2(a) has the following c-regions:

$$\begin{aligned}
\mathfrak{r}_1 &= (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing) & \mathfrak{r}_2 &= (\varnothing, \varnothing, \varnothing, \{\mathtt{e}\}, \varnothing)\\
\mathfrak{r}_3 &= (\varnothing, \varnothing, \varnothing, \{\mathtt{f}\}, \varnothing) & \mathfrak{r}_4 &= (\varnothing, \varnothing, \varnothing, \{\mathtt{e}, \mathtt{f}\}, \varnothing)\\
\mathfrak{r}_5 &= (\{s_{init}, s\}, \varnothing, \varnothing, \varnothing, \varnothing) & \mathfrak{r}_6 &= (\{s_{init}, s\}, \varnothing, \varnothing, \varnothing, \{\mathtt{e}\})\\
\mathfrak{r}_7 &= (\{s_{init}, s\}, \varnothing, \varnothing, \varnothing, \{\mathtt{f}\}) & \mathfrak{r}_8 &= (\{s_{init}, s\}, \varnothing, \varnothing, \varnothing, \{\mathtt{e}, \mathtt{f}\})\\
\mathfrak{r}_9 &= (\{s_{init}\}, \varnothing, \{\mathtt{f}\}, \varnothing, \varnothing\}) & \mathfrak{r}_{10} &= (\{s_{init}\}, \varnothing, \{\mathtt{f}\}, \varnothing, \{\mathtt{e}\})\\
\mathfrak{r}_{11} &= (\{s_{init}\}, \varnothing, \{\mathtt{e}\}, \varnothing, \varnothing\}) & \mathfrak{r}_{12} &= (\{s_{init}\}, \varnothing, \{\mathtt{e}\}, \varnothing, \{\mathtt{f}\})\\
\mathfrak{r}_{13} &= (\{s\}, \{\mathtt{f}\}, \varnothing, \varnothing, \varnothing\}) & \mathfrak{r}_{14} &= (\{s\}, \{\mathtt{f}\}, \varnothing, \{\mathtt{e}\}, \varnothing\})\\
\mathfrak{r}_{15} &= (\{s\}, \{\mathtt{e}\}, \varnothing, \varnothing, \varnothing\}) & \mathfrak{r}_{16} &= (\{s\}, \{\mathtt{e}\}, \varnothing, \{\mathtt{f}\}, \varnothing)\,.
\end{aligned}$$

In the rest of this section, we discuss and prove properties of c-regions which will subsequently be needed to solve the synthesis problem for ENCL-systems.

Trivial c-regions. A c-region $\mathfrak{r}$ is *trivial* if $\|\mathfrak{r}\| = \varnothing$ or $\|\mathfrak{r}\| = S$; otherwise it is *non-trivial*. For example, the step transition system shown in Figure 2(a) has eight trivial c-regions ($\mathfrak{r}_1, \ldots, \mathfrak{r}_8$) and eight non-trivial c-regions ($\mathfrak{r}_9, \ldots, \mathfrak{r}_{16}$). Note that only non-trivial c-regions will be used in the synthesis procedure.

Proposition 2. *If $\mathfrak{r}$ is a trivial c-region then ${}^\bullet\mathfrak{r} = \mathfrak{r}^\bullet = \varnothing$.*

Proof. Follows from Definition 4(5,6) and TSYS2. □

Proposition 3. *If $\mathfrak{r}$ is a c-region then the* complement *of $\mathfrak{r}$, defined as $\overline{\mathfrak{r}} \overset{\text{df}}{=} (S \setminus \|\mathfrak{r}\|, \mathfrak{r}^\bullet, {}^\bullet\mathfrak{r}, {}^\blacktriangleleft\mathfrak{r}, {}^\blacklozenge\mathfrak{r})$, is also a c-region.*

Proof. Follows directly from Definition 4. □

The set of all non-trivial c-regions will be denoted by $\text{REG}_{\mathfrak{ts}}$ and, for every state $s \in S$, we will denote by REG_s the set of all the non-trivial c-regions containing s, $\text{REG}_s \overset{\text{df}}{=} \{\mathfrak{r} \in \text{REG}_{\mathfrak{ts}} \mid s \in \|\mathfrak{r}\|\}$. For the example in Figure 2(a), we have $\text{REG}_{s_{init}} = \{\mathfrak{r}_9, \mathfrak{r}_{10}, \mathfrak{r}_{11}, \mathfrak{r}_{12}\}$ and $\overline{\mathfrak{r}_{12}} = \mathfrak{r}_{16}$.

Lattices of c-regions. We call two c-regions, $\mathfrak{r}$ and $\mathfrak{r}'$, *compatible* if it is the case that $\|\mathfrak{r}\| = \|\mathfrak{r}'\|$, ${}^\bullet\mathfrak{r} = {}^\bullet\mathfrak{r}'$ and $\mathfrak{r}^\bullet = \mathfrak{r}'^\bullet$. We denote this by $\mathfrak{r} \approx \mathfrak{r}'$. For two compatible c-regions, $\mathfrak{r}$ and $\mathfrak{r}'$, we define their *union* and *intersection*, in the following way:

$$\mathfrak{r} \cup \mathfrak{r}' \overset{\mathrm{df}}{=} (\|\mathfrak{r}\|, {}^\bullet\mathfrak{r}, \mathfrak{r}^\bullet, {}^\blacklozenge\mathfrak{r} \cup {}^\blacklozenge\mathfrak{r}', {}^\blacktriangleleft\mathfrak{r} \cup {}^\blacktriangleleft\mathfrak{r}') \text{ and } \mathfrak{r} \cap \mathfrak{r}' \overset{\mathrm{df}}{=} (\|\mathfrak{r}\|, {}^\bullet\mathfrak{r}, \mathfrak{r}^\bullet, {}^\blacklozenge\mathfrak{r} \cap {}^\blacklozenge\mathfrak{r}', {}^\blacktriangleleft\mathfrak{r} \cap {}^\blacktriangleleft\mathfrak{r}') .$$

Moreover, we denote $\mathfrak{r} \preceq \mathfrak{r}'$ whenever ${}^\blacklozenge\mathfrak{r} \subseteq {}^\blacklozenge\mathfrak{r}'$ and ${}^\blacktriangleleft\mathfrak{r} \subseteq {}^\blacktriangleleft\mathfrak{r}'$. For the example in Figure 2(a), we have $\mathfrak{r}_1 \approx \mathfrak{r}_2 \approx \mathfrak{r}_3 \approx \mathfrak{r}_4$, $\mathfrak{r}_2 \cup \mathfrak{r}_3 = \mathfrak{r}_4$ and $\mathfrak{r}_{15} \preceq \mathfrak{r}_{16}$.

Proposition 4. *If $\mathfrak{r}$ is a c-region, and $inh \subseteq {}^\blacklozenge\mathfrak{r}$ and $act \subseteq {}^\blacktriangleleft\mathfrak{r}$ are two sets of events, then $(\|\mathfrak{r}\|, {}^\bullet\mathfrak{r}, \mathfrak{r}^\bullet, inh, act)$ is also a c-region.*

Proof. Follows directly from Definition 4. □

Proposition 5. *If $\mathfrak{r}$ and $\mathfrak{r}'$ are compatible c-regions, then $\mathfrak{r} \cup \mathfrak{r}'$ and $\mathfrak{r} \cap \mathfrak{r}'$ are also c-regions.*

Proof. The first part follows directly from Definition 4, and the second from Proposition 4. □

Given a c-region $\mathfrak{r}$, the equivalence class of c-regions compatible with $\mathfrak{r}$, denoted by $[\mathfrak{r}]_\approx$, forms a complete lattice w.r.t. the partial order $\preceq$ and the operations $\cup$ (join) and $\cap$ (meet). The $\preceq$-*minimal* and $\preceq$-*maximal* c-regions it contains are given respectively by:

$$(\|\mathfrak{r}\|, {}^\bullet\mathfrak{r}, \mathfrak{r}^\bullet, \varnothing, \varnothing) \qquad \text{and} \qquad (\|\mathfrak{r}\|, {}^\bullet\mathfrak{r}, \mathfrak{r}^\bullet, \bigcup_{\mathfrak{r}' \in [\mathfrak{r}]_\approx} {}^\blacklozenge\mathfrak{r}', \bigcup_{\mathfrak{r}' \in [\mathfrak{r}]_\approx} {}^\blacktriangleleft\mathfrak{r}') .$$

The step transition system in Figure 2(a) has six $\preceq$-minimal c-regions ($\mathfrak{r}_1$, $\mathfrak{r}_5$, $\mathfrak{r}_9$, $\mathfrak{r}_{11}$, $\mathfrak{r}_{13}$ and $\mathfrak{r}_{15}$) and six $\preceq$-maximal c-regions ($\mathfrak{r}_4$, $\mathfrak{r}_8$, $\mathfrak{r}_{10}$, $\mathfrak{r}_{12}$, $\mathfrak{r}_{14}$ and $\mathfrak{r}_{16}$).

We feel that the algebraic properties enjoyed by sets of compatible c-regions will be useful in the synthesis procedure aimed at constructing optimal ENCL-systems. We will come back to this issue later on.

Relating regions and events. Given an event $e \in \mathcal{E}_{\mathfrak{ts}}$, its sets of *pre-c-regions*, ${}^\circ e$, *post-c-regions*, e°, *inh-c-regions*, ${}^\Diamond e$, and *act-c-regions*, ${}^\triangleleft e$, are respectively defined as:

$$\begin{aligned} {}^\circ e &\overset{\mathrm{df}}{=} \{\mathfrak{r} \in \mathrm{REG}_{\mathfrak{ts}} \mid e \in \mathfrak{r}^\bullet\} & e^\circ &\overset{\mathrm{df}}{=} \{\mathfrak{r} \in \mathrm{REG}_{\mathfrak{ts}} \mid e \in {}^\bullet\mathfrak{r}\} \\ {}^\Diamond e &\overset{\mathrm{df}}{=} \{\mathfrak{r} \in \mathrm{REG}_{\mathfrak{ts}} \mid e \in {}^\blacklozenge\mathfrak{r}\} & {}^\triangleleft e &\overset{\mathrm{df}}{=} \{\mathfrak{r} \in \mathrm{REG}_{\mathfrak{ts}} \mid e \in {}^\blacktriangleleft\mathfrak{r}\} . \end{aligned}$$

Moreover, ${}^\circ e^\circ \overset{\mathrm{df}}{=} {}^\circ e \cup e^\circ$, ${}^\circ e^\triangleleft \overset{\mathrm{df}}{=} {}^\circ e \cup {}^\triangleleft e$ and ${}^\Diamond e^\circ \overset{\mathrm{df}}{=} e^\circ \cup {}^\Diamond e$. All these notations can be applied to sets of events by taking the union of sets of regions defined for the individual events. For the step transition system in Figure 2(a), we have ${}^\circ\mathtt{e} = \{\mathfrak{r}_{11}, \mathfrak{r}_{12}\}$ and ${}^\Diamond\mathtt{f} = \{\mathfrak{r}_{16}\}$.

Proposition 6. *If $s \xrightarrow{u} s'$ is a transition of $\mathfrak{ts}$, then:*

1. $\mathfrak{r} \in {}^{\circ}u$ *implies* $s \in \|\mathfrak{r}\|$ *and* $s' \notin \|\mathfrak{r}\|$.
2. $\mathfrak{r} \in u^{\circ}$ *implies* $s \notin \|\mathfrak{r}\|$ *and* $s' \in \|\mathfrak{r}\|$.
3. $\mathfrak{r} \in {}^{\diamond}u$ *implies* $s \notin \|\mathfrak{r}\|$.
4. $\mathfrak{r} \in {}^{\triangleleft}u$ *implies* $s \in \|\mathfrak{r}\|$.

Proof. Follows directly from the definitions of ${}^{\circ}u$, u°, ${}^{\diamond}u$ and ${}^{\triangleleft}u$ as well as Definition 4(3,4,5,6). □

The sets of pre-c-regions and post-c-regions of events in an executed step are mutually disjoint. Moreover, they can be 'calculated' using the c-regions associated with the source and target states.

Proposition 7. *If $s \xrightarrow{u} s'$ is a transition of $\mathfrak{ts}$, then:*

1. ${}^{\circ}e \cap {}^{\circ}f = \varnothing$ *and* $e^{\circ} \cap f^{\circ} = \varnothing$, *for all distinct* $e, f \in u$.
2. ${}^{\circ}u \cap u^{\circ} = \varnothing$.
3. ${}^{\circ}u = \textsc{Reg}_s \setminus \textsc{Reg}_{s'}$ *and* $u^{\circ} = \textsc{Reg}_{s'} \setminus \textsc{Reg}_s$.

Proof. (1) Suppose that $\mathfrak{r} \in {}^{\circ}e \cap {}^{\circ}f$, i.e., $e, f \in \mathfrak{r}^{\bullet}$. This means, by Definition 4(5), that $s \in \|\mathfrak{r}\|$ and $s' \notin \|\mathfrak{r}\|$. Thus, by Definition 4(1), $|u \cap \mathfrak{r}^{\bullet}| = 1$, a contradiction with $e, f \in u \cap \mathfrak{r}^{\bullet}$. The second part can be shown in a similar way.

(2) Suppose that $\mathfrak{r} \in {}^{\circ}u \cap u^{\circ}$. Then, by Proposition 6(1,2), $s \in \|\mathfrak{r}\|$ and $s \notin \|\mathfrak{r}\|$, a contradiction.

(3) We only show that $\textsc{Reg}_s \setminus \textsc{Reg}_{s'} = {}^{\circ}u$, as the second part can be shown in a similar way. By Proposition 6, ${}^{\circ}u \subseteq \textsc{Reg}_s$ and ${}^{\circ}u \cap \textsc{Reg}_{s'} = \varnothing$. Hence ${}^{\circ}u \subseteq \textsc{Reg}_s \setminus \textsc{Reg}_{s'}$. Suppose that $\mathfrak{r} \in \textsc{Reg}_s \setminus \textsc{Reg}_{s'}$, which implies that $s \in \|\mathfrak{r}\|$ and $s' \notin \|\mathfrak{r}\|$. Hence, by Definition 4(1) and $s \xrightarrow{u} s'$, $u \cap \mathfrak{r}^{\bullet} \neq \varnothing$. Thus $\mathfrak{r} \in {}^{\circ}u$ and so $\textsc{Reg}_s \setminus \textsc{Reg}_{s'} \subseteq {}^{\circ}u$. Consequently, $\textsc{Reg}_s \setminus \textsc{Reg}_{s'} = {}^{\circ}u$. □

The next two propositions provide a useful characterisation of inh-c-regions and act-c-regions of an event in terms of transitions involving this event. For example, if $\mathfrak{r}$ is an inh-c-region of event e, then no transition involving e lies completely within $\mathfrak{r}$. In what follows, for an event e and a c-region $\mathfrak{r}$, we denote $\mathcal{B}^{e}_{\mathfrak{r}} \overset{\mathrm{df}}{=} \{(s, u, s') \in T_e \mid s, s' \in \|\mathfrak{r}\|\}$ to be the set of all transitions involving e which are *buried* in $\mathfrak{r}$, i.e., their source and target states belong to $\|\mathfrak{r}\|$.

Proposition 8. *If $e \in \mathcal{E}_{\mathfrak{ts}}$ and $\mathfrak{r} \in {}^{\diamond}e$, then one of the following holds:*

1. $\mathcal{B}^{e}_{\mathfrak{r}} = \varnothing$, $\mathcal{B}^{e}_{\overline{\mathfrak{r}}} \neq \varnothing$ *and* $\mathfrak{r} \notin {}^{\circ}u$, *for all* $u \in U_e$.
2. $\mathcal{B}^{e}_{\mathfrak{r}} = \varnothing$, $e \notin {}^{\bullet}\mathfrak{r}$ *and* $\mathfrak{r} \in u^{\circ} \setminus {}^{\circ}u$, *for all* $u \in U_e$.

Proof. Suppose that $(s, u, s') \in T_e$. From $\mathfrak{r} \in {}^{\diamond}e \subseteq {}^{\diamond}u$ and Proposition 6(3), we have that $s \notin \|\mathfrak{r}\|$. Hence $\mathcal{B}^{e}_{\mathfrak{r}} = \varnothing$ and $\mathfrak{r} \notin {}^{\circ}u$, for all $u \in U_e$. We will now show that either $\mathcal{B}^{e}_{\overline{\mathfrak{r}}} \neq \varnothing$, or that $e \notin {}^{\bullet}\mathfrak{r}$ and $\mathfrak{r} \in u^{\circ}$, for all $u \in U_e$.

Suppose that $\mathcal{B}^{e}_{\overline{\mathfrak{r}}} = \varnothing$. We first observe that $e \notin {}^{\bullet}\mathfrak{r}$ since it follows directly from $e \in {}^{\blacklozenge}\mathfrak{r}$ (as $\mathfrak{r} \in {}^{\diamond}e$) and Definition 4(7). What remains to be shown is that if $(s, u, s') \in T_e$ then $\mathfrak{r} \in u^{\circ}$. We already know that $s \notin \|\mathfrak{r}\|$. Moreover, since $\mathcal{B}^{e}_{\overline{\mathfrak{r}}} = \varnothing$, we have $s' \in \|\mathfrak{r}\|$. This means, by Definition 4(2), that $|u \cap {}^{\bullet}\mathfrak{r}| = 1$. Hence there is $f \in u$ such that $f \in {}^{\bullet}\mathfrak{r}$, and so $\mathfrak{r} \in f^{\circ} \subseteq u^{\circ}$. □

Proposition 9. *If $e \in \mathcal{E}_{\mathfrak{ts}}$ and $\mathfrak{r} \in {}^{\triangleleft}e$, then one of the following holds:*

1. *$\mathcal{B}^{e}_{\overline{\mathfrak{r}}} = \varnothing$, $\mathcal{B}^{e}_{\mathfrak{r}} \neq \varnothing$ and $\mathfrak{r} \notin u^{\circ}$, for all $u \in U_e$.*
2. *$\mathcal{B}^{\underline{e}}_{\overline{\mathfrak{r}}} = \varnothing$, $e \notin \mathfrak{r}^{\bullet}$ and $\mathfrak{r} \in {}^{\circ}u \setminus u^{\circ}$, for all $u \in U_e$.*

Proof. Similar to that of Proposition 8. □

It is easy to show that a step can be executed at a state only if the inh-c-regions of the former do not comprise the latter, and the act-c-regions do.

Proposition 10. *If $s \xrightarrow{u} s'$ is a transition of $\mathfrak{ts}$, then ${}^{\lozenge}u \cap \text{REG}_s = \varnothing$ and ${}^{\triangleleft}u \subseteq \text{REG}_s$.*

Proof. Suppose that $\mathfrak{r} \in {}^{\lozenge}u \cap \text{REG}_s \neq \varnothing$. Then from $\mathfrak{r} \in {}^{\lozenge}u$ and Proposition 6(3) we have that $s \notin \|\mathfrak{r}\|$ which contradicts $\mathfrak{r} \in \text{REG}_s$. Suppose now that $\mathfrak{r} \in {}^{\triangleleft}u$. From Proposition 6(4) we have that $s \in \|\mathfrak{r}\|$, and so $\mathfrak{r} \in \text{REG}_s$. □

Proposition 11. *If $e \in \mathcal{E}_{\mathfrak{ts}}$, then ${}^{\circ}e^{\circ} \cap ({}^{\lozenge}e \cup {}^{\triangleleft}e) = \varnothing$.*

Proof. Suppose that $\mathfrak{r} \in e^{\circ} \cap {}^{\lozenge}e \neq \varnothing$. Then $e \in {}^{\bullet}\mathfrak{r} \cap {}^{\blacklozenge}\mathfrak{r} \neq \varnothing$, contradicting Definition 4(7).

Suppose now that $\mathfrak{r} \in {}^{\circ}e \cap {}^{\lozenge}e \neq \varnothing$. By Proposition 8, one of the following two cases holds:

Case 1: There is $(s, u, s') \in T_e$ such that $s, s' \notin \|\mathfrak{r}\|$. By $\mathfrak{r} \in {}^{\circ}e$, we have that $\mathfrak{r} \in {}^{\circ}u$, and so from Proposition 6 it follows that $s \in \|\mathfrak{r}\|$ and $s' \notin \|\mathfrak{r}\|$, a contradiction.
Case 2: $e \notin {}^{\bullet}\mathfrak{r}$ and $\mathfrak{r} \in u^{\circ}$ for some $u \in U_e \neq \varnothing$. Then $\mathfrak{r} \notin e^{\circ}$ and there is $(s, u, s') \in T_e$ such that $s \notin \|\mathfrak{r}\|$ and $s' \in \|\mathfrak{r}\|$. On the other hand, by $\mathfrak{r} \in {}^{\circ}e \subseteq {}^{\circ}u$ and Proposition 6, we have $s \in \|\mathfrak{r}\|$ and $s' \notin \|\mathfrak{r}\|$, a contradiction.

Hence ${}^{\circ}e^{\circ} \cap {}^{\lozenge}e = \varnothing$, and ${}^{\circ}e^{\circ} \cap {}^{\triangleleft}e = \varnothing$ can be shown in a similar way. □

To characterise transition systems generated by ENCL-systems, we will need the set of all *potential steps* $\mathbb{U}_{\mathfrak{ts}}$ of $\mathfrak{ts}$, given by:

$$\mathbb{U}_{\mathfrak{ts}} \overset{\text{df}}{=} \{u \subseteq \mathcal{E}_{\mathfrak{ts}} \mid u \neq \varnothing \ \wedge \ \forall e, f \in u : e \neq f \Rightarrow {}^{\circ}e^{\circ} \cap {}^{\circ}f^{\circ} = \varnothing\} \ .$$

Proposition 12. *If $s \xrightarrow{u} s'$ is a transition of $\mathfrak{ts}$, then $u \in \mathbb{U}_{\mathfrak{ts}}$.*

Proof. Follows from TSYS2 and Proposition 7(1,2). □

Thin transition systems. In general, a c-region $\mathfrak{r}$ cannot be identified only by its set of states $\|\mathfrak{r}\|$; in other words, ${}^{\bullet}\mathfrak{r}$, $\mathfrak{r}^{\bullet}$, ${}^{\blacklozenge}\mathfrak{r}$ and ${}^{\blacktriangleleft}\mathfrak{r}$ may not be recoverable from $\|\mathfrak{r}\|$. However, if the transition system is *thin*, i.e., for every event $e \in \mathcal{E}_{\mathfrak{ts}}$ we have that $\{e\} \in U_e$, then different c-regions with the same sets *inh* and *act* are based on different sets of states.

Proposition 13 ([4]). *If $\mathfrak{ts}$ is thin and $\mathfrak{r} \neq \mathfrak{r}'$ are c-regions such that ${}^{\blacklozenge}\mathfrak{r} = {}^{\blacklozenge}\mathfrak{r}'$ and ${}^{\blacktriangleleft}\mathfrak{r} = {}^{\blacktriangleleft}\mathfrak{r}'$, then $\|\mathfrak{r}\| \neq \|\mathfrak{r}'\|$.*

4 Transition Systems of ENCL-Systems

We now can present a complete characterisation of the transition systems generated by ENCL-systems.

Definition 5 (ENCL-transition system). *A step transition system* $\mathfrak{ts} = (S, T, s_{init})$ *is an* ENCL-transition system *if it satisfies the following axioms:*

AXIOM1 *For every* $s \in S \setminus \{s_{init}\}$, *there are* $(s_0, u_0, s_1), \ldots, (s_{n-1}, u_{n-1}, s_n) \in T$ *such that* $s_0 = s_{init}$ *and* $s_n = s$.

AXIOM2 *For every event* $e \in \mathcal{E}_{\mathfrak{ts}}$, *both* $^\circ e$ *and* e° *are non-empty.*

AXIOM3 *For all states* $s, s' \in S$, *if* $\mathrm{REG}_s = \mathrm{REG}_{s'}$ *then* $s = s'$.

AXIOM4 *If* $s \in S$ *and* $u \in \mathbb{U}_{\mathfrak{ts}}$ *are such that*
- $^\circ u^\triangleleft \subseteq \mathrm{REG}_s$ *and* $^\Diamond u^\circ \cap \mathrm{REG}_s = \varnothing$ *and*
- *there is no step* $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{ts}}$ *with the event* e *satisfying* $\mathfrak{L}(e) \in \mathfrak{L}(u)$, $^\circ e^\triangleleft \subseteq \mathrm{REG}_s$ *and* $^\Diamond e^\circ \cap \mathrm{REG}_s = \varnothing$,

then we have $s \xrightarrow{u}$.

AXIOM5 *If* $s \xrightarrow{u}$ *then there is no step* $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{ts}}$ *with the event* e *satisfying* $\mathfrak{L}(e) \in \mathfrak{L}(u)$, $^\circ e^\triangleleft \subseteq \mathrm{REG}_s$ *and* $^\Diamond e^\circ \cap \mathrm{REG}_s = \varnothing$.

In the above, AXIOM1 implies that all the states in $\mathfrak{ts}$ are reachable from the initial state. AXIOM2 will ensure that every event in a synthesised ENCL-system will have at least one input condition and at least one output condition. AXIOM3 was used for other transition systems as well, and is usually called the *state separation property* [16,20], and it guarantees that $\mathfrak{ts}$ is deterministic. AXIOM4 is a variation of the *forward closure property* [20] or the *event/state separation property* [16]. AXIOM5 ensures that every step in a transition system is indeed a maximal step w.r.t. localities of the events it comprises.

Proposition 14. *If* $s \xrightarrow{u} s'$ *and* $s \xrightarrow{u} s''$, *then* $s' = s''$.

Proof. Follows from Proposition 7(3) and AXIOM3. □

The construction of a step transition system for a given ENCL-system is straightforward.

Definition 6 (from net system to transition system). *The* transition system generated *by an* ENCL-*system* $\mathfrak{encl}$ *is* $\mathfrak{ts}_{\mathfrak{encl}} \overset{\mathrm{df}}{=} (C_{\mathfrak{encl}}, \rightarrow_{\mathfrak{encl}}, c_{init})$, *where* c_{init} *is the initial case of* $\mathfrak{encl}$.

Theorem 1. $\mathfrak{ts}_{\mathfrak{encl}}$ *is an* ENCL-*transition system.*

Proof. See the Appendix. □

5 Solving the Synthesis Problem

The translation from ENCL-transition systems to ENCL-systems is based on the pre-, post-, inh- and act-c-regions of the events appearing in a transition system.

Definition 7 (from transition system to net system). *The net system* associated *with an* ENCL-*transition system* $\mathfrak{ts} = (S, T, s_{init})$ *is:*

$$\mathfrak{encl}_{\mathfrak{ts}} \overset{\text{df}}{=} (\text{REG}_{\mathfrak{ts}}, \mathcal{E}_{\mathfrak{ts}}, F_{\mathfrak{ts}}, I_{\mathfrak{ts}}, A_{\mathfrak{ts}}, \text{REG}_{s_{init}}) ,$$

where $F_{\mathfrak{ts}}$, $I_{\mathfrak{ts}}$ *and* $A_{\mathfrak{ts}}$ *are defined thus:*

$$\left.\begin{array}{l} F_{\mathfrak{ts}} \overset{\text{df}}{=} \{(\mathfrak{r}, e) \in \text{REG}_{\mathfrak{ts}} \times \mathcal{E}_{\mathfrak{ts}} \mid \mathfrak{r} \in {}^{\circ}e\} \;\cup\; \{(e, \mathfrak{r}) \in \mathcal{E}_{\mathfrak{ts}} \times \text{REG}_{\mathfrak{ts}} \mid \mathfrak{r} \in e^{\circ}\} \\ I_{\mathfrak{ts}} \overset{\text{df}}{=} \{(\mathfrak{r}, e) \in \text{REG}_{\mathfrak{ts}} \times \mathcal{E}_{\mathfrak{ts}} \mid \mathfrak{r} \in {}^{\diamond}e\} \\ A_{\mathfrak{ts}} \overset{\text{df}}{=} \{(\mathfrak{r}, e) \in \text{REG}_{\mathfrak{ts}} \times \mathcal{E}_{\mathfrak{ts}} \mid \mathfrak{r} \in {}^{\triangleleft}e\} . \end{array}\right\} \quad (1)$$

Proposition 15. *For every* $e \in \mathcal{E}_{\mathfrak{ts}}$*, we have* ${}^{\circ}e = {}^{\bullet}e$*,* $e^{\circ} = e^{\bullet}$*,* ${}^{\diamond}e = {}^{\blacklozenge}e$ *and* ${}^{\triangleleft}e = {}^{\blacktriangleleft}e$*.*

Proof. Follows directly from the definition of $\mathfrak{encl}_{\mathfrak{ts}}$. □

Figure 3 shows the ENCL-system associated with the step transition system shown in Figure 2(a). It is, clearly, not the net system shown in Figure 2(b), as it contains twice as many conditions as well as four context arcs which were not present there. This is not unusual as the above construction produces nets which are saturated with conditions as well as context arcs. In fact, the whole construction would still work if we restricted ourselves to the $\preceq$-maximal non-trivial c-regions, similarly as it has been done in [21] for EN-systems with inhibitor arcs. But the resulting ENCL-system would still not be as that shown in Figure 2(b). In fact, the latter would be re-constructed if we took all the $\preceq$-minimal non-trivial c-regions of the step transition system shown in Figure 2(a). However, taking only the $\preceq$-minimal c-regions would not work in the general case (the ENCL-transition system shown in Figure 4(c) provides a suitable counterexample), and that it is possible to use them in this case is due to the maximally concurrent execution rule which underpins ENCL-systems. What this example implies is that in order to synthesise an optimal net (for example, from the point of view of the number of conditions and/or context arcs), it is a good idea to look at the whole spectrum of c-regions arranged in the lattices of compatible c-regions (and, in any case, never use two different c-regions, $\mathfrak{r}$ and $\mathfrak{r}'$, such that $\mathfrak{r} \preceq \mathfrak{r}'$).

Theorem 2. $\mathfrak{encl}_{\mathfrak{ts}}$ *is an* ENCL-*system.*

Proof. All one needs to observe is that, for every $e \in \mathcal{E}_{\mathfrak{ts}}$, it is the case that: ${}^{\bullet}e \neq \varnothing \neq e^{\bullet}$, which follows from AXIOM2 and Proposition 15; ${}^{\bullet}e \cap e^{\bullet} = \varnothing$, which follows from Propositions 7(2) and 15; and ${}^{\bullet}e^{\bullet} \cap ({}^{\blacklozenge}e \cup {}^{\blacktriangleleft}e) = \varnothing$, which follows from Propositions 11 and 15. □

We finally show that the ENCL-system associated with an ENCL-transition system $\mathfrak{ts}$ generates a transition system which is isomorphic to $\mathfrak{ts}$.

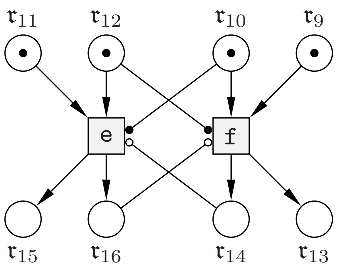


Fig. 3. ENCL-system synthesised from the ENCL-transition system in Figure 2(a)

Proposition 16. *Let $\mathfrak{ts} = (S, T, s_{init})$ be an* ENCL*-transition system and*

$$\mathfrak{encl} = \mathfrak{encl}_{\mathfrak{ts}} = (\mathrm{REG}_{\mathfrak{ts}}, \mathcal{E}_{\mathfrak{ts}}, F_{\mathfrak{ts}}, I_{\mathfrak{ts}}, A_{\mathfrak{ts}}, \mathrm{REG}_{s_{init}}) = (B, E, F, I, A, c_{init})$$

be the ENCL*-system associated with it.*

1. $C_{\mathfrak{encl}} = \{\mathrm{REG}_s \mid s \in S\}$.
2. $\rightarrow_{\mathfrak{encl}} = \{(\mathrm{REG}_s, u, \mathrm{REG}_{s'}) \mid (s, u, s') \in T\}$.

Proof. Note that from the definition of $C_{\mathfrak{encl}}$, every $c \in C_{\mathfrak{encl}}$ is reachable from c_{init} in $\mathfrak{encl}$; and that from AXIOM1, every $s \in S$ is reachable from s_{init} in $\mathfrak{ts}$.

We first show that if $c \xrightarrow{u}_{\mathfrak{encl}} c'$ and $c = \mathrm{REG}_s$, for some $s \in S$, then there is $s' \in S$ such that $s \xrightarrow{u} s'$ and $c' = \mathrm{REG}_{s'}$. By $c \xrightarrow{u}_{\mathfrak{encl}} c'$, $u \in \mathbb{U}_{\mathfrak{encl}}$ is a step such that ${}^\bullet u^\blacktriangleleft \subseteq c$ and ${}^\blacklozenge u^\bullet \cap c = \varnothing$, and there is no step $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{encl}}$ satisfying $\mathfrak{L}(e) \in \mathfrak{L}(u)$ and ${}^\bullet e^\blacktriangleleft \subseteq c$ and ${}^\blacklozenge e^\bullet \cap c = \varnothing$. Moreover, $c' = (c \setminus {}^\bullet u) \cup u^\bullet$.

Hence, by Proposition 15 and AXIOM4, $u \in \mathbb{U}_{\mathfrak{ts}}$ and $s \xrightarrow{u} s'$, for some $s' \in S$. Then, by Proposition 7(3), $\mathrm{REG}_{s'} = (\mathrm{REG}_s \setminus {}^\circ u) \cup u^\circ$. At the same time, we have $c' = (c \setminus {}^\bullet u) \cup u^\bullet$. Hence, by Proposition 15 and $c = \mathrm{REG}_s$, we have that $c' = \mathrm{REG}_{s'}$.

As a result, we have shown (note that $c_{init} = \mathrm{REG}_{s_{init}} \in \{\mathrm{REG}_s \mid s \in S\}$) that

$$\begin{array}{lcl} C_{\mathfrak{encl}} & \subseteq & \{\mathrm{REG}_s \mid s \in S\} \\ \rightarrow_{\mathfrak{encl}} & \subseteq & \{(\mathrm{REG}_s, u, \mathrm{REG}_{s'}) \mid (s, u, s') \in T\}\ . \end{array}$$

We now prove the reverse inclusions. By definition, $\mathrm{REG}_{s_{init}} \in C_{\mathfrak{encl}}$. It is enough to show that if $s \xrightarrow{u} s'$ and $\mathrm{REG}_s \in C_{\mathfrak{encl}}$, then $\mathrm{REG}_{s'} \in C_{\mathfrak{encl}}$ and $\mathrm{REG}_s \xrightarrow{u}_{\mathfrak{encl}} \mathrm{REG}_{s'}$. By AXIOM5 and Propositions 7(3), 12, 10 and 15, u is a valid step in $\mathfrak{encl}$ which is enabled at the case REG_s. So, there is a case c' such that $\mathrm{REG}_s \xrightarrow{u}_{\mathfrak{encl}} c'$ and $c' = (\mathrm{REG}_s \setminus {}^\bullet u) \cup u^\bullet$. From Propositions 7(3) and 15 we have that $c' = \mathrm{REG}_{s'}$. Hence we obtain that $\mathrm{REG}_s \xrightarrow{u}_{\mathfrak{encl}} \mathrm{REG}_{s'}$ and so also $\mathrm{REG}_{s'} \in C_{\mathfrak{encl}}$. $\square$

Theorem 3. *Let $\mathfrak{ts} = (S, T, s_{init})$ be an* ENCL*-transition system and $\mathfrak{encl} = \mathfrak{encl}_{\mathfrak{ts}}$ be the* ENCL*-system associated with it. Then $\mathfrak{ts}_{\mathfrak{encl}}$ is isomorphic to $\mathfrak{ts}$.*

Proof. Let $\psi : S \to C_{\mathfrak{encl}}$ be a mapping given by $\psi(s) = \text{Reg}_s$, for all $s \in S$ (note that, by Proposition 16(1), ψ is well-defined). We will show that ψ is an isomorphism for $\mathfrak{ts}$ and $\mathfrak{ts}_{\mathfrak{encl}}$.

Note that $\psi(s_{init}) = \text{Reg}_{s_{init}}$. By Proposition 16(1), ψ is onto. Moreover, by Axiom3, it is injective. Hence ψ is a bijection. We then observe that, by Proposition 16(2), we have $(s, u, s') \in T$ if and only if $(\psi(s), u, \psi(s')) \in \to_{\mathfrak{encl}}$. Hence ψ is an isomorphism for $\mathfrak{ts}$ and $\mathfrak{ts}_{\mathfrak{encl}}$. $\square$

Figure 4 shows two further examples of the synthesis of ENCL-systems. The first one, in Figure 4(a,b), illustrates a conflict between two events, and the synthesised ENCL-system utilises two $\preceq$-minimal c-regions, $\mathfrak{r} = (\{s_{init}\}, \varnothing, \{\mathtt{e}, \mathtt{f}\}, \varnothing, \varnothing)$ for the upper condition, and its complement $\overline{\mathfrak{r}}$ for the lower one. The second example, in Figure 4(c,d), exemplifies a situation when a correct solution has been obtained without using only $\preceq$-maximal c-regions. However, an attempt to use only $\preceq$-minimal c-regions would fail, as the resulting ENCL-system (shown in Figure 5(a)) allows one to execute the step sequence $\{\mathtt{e}\}\{\mathtt{f}\}$ which is impossible in the original transition system. Moreover, Figure 5(b) shows a correct synthesis solution based solely on $\preceq$-maximal c-regions. When compared with that in Figure 4(d) it looks less attractive since the latter uses fewer context arcs. It should already be clear that to synthesise 'optimal' ENCL-systems it will, in general, be necessary to use a mix of various kinds of c-regions, and the development of suitable algorithms is an interesting and important topic for further research.

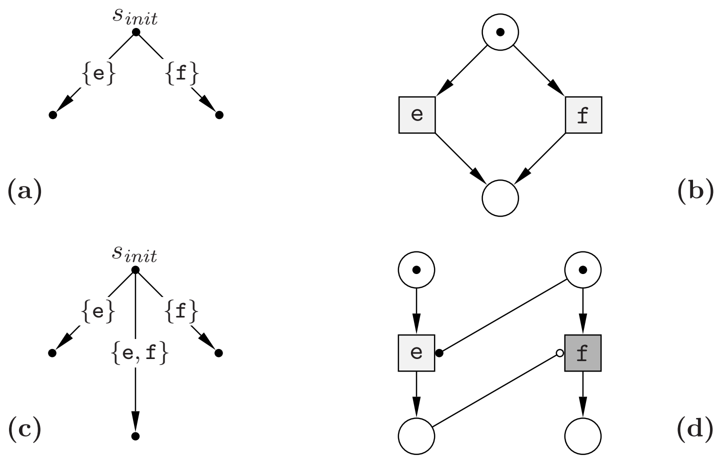


Fig. 4. A transition system with co-located events $\mathtt{e}$ and $\mathtt{f}$ **(a)**, and a corresponding ENCL-system **(b)**; and a transition system with differently located events $\mathtt{e}$ and $\mathtt{f}$ **(c)**, and a corresponding ENCL-system **(d)**

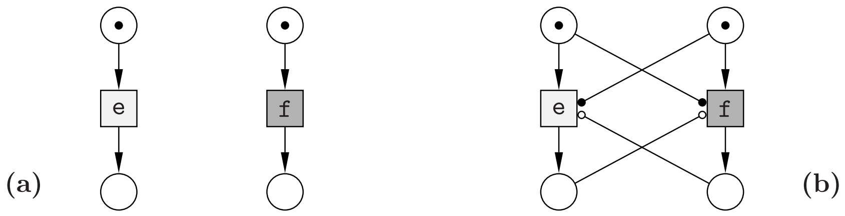


Fig. 5. ENCL-system synthesised from the transition system in Figure 4(c) using only $\preceq$-minimal non-trivial c-regions **(a)**, and only $\preceq$-maximal non-trivial c-regions **(b)**

6 Concluding Remarks

In this paper, we solved the synthesis problem for EN-systems with context arcs and localities, by following the standard approach in which key relationships between a Petri net and its transition system are established via the notion of a region. Moreover, in order to obtain a satisfactory solution, we augmented the standard notion of a region with some additional information, leading to the notion of a c-region. We then defined, and showed consistency of, two behaviour preserving translations between ENCL-systems and their transition systems.

Throughout this paper it has always been assumed that events' localities are known in advance. In particular, this information was present in the input to the synthesis problem. However, one might prefer to work only with step transition systems, and determine the localities of events during the synthesis procedure (perhaps choosing an 'optimal' option). This could, of course, be done by considering in turn all possibilities for the locality mapping $\mathfrak{L}$. Unfortunately, such an approach would be hardly satisfactory as there are $B_{|\mathcal{E}_{\mathfrak{ts}}|}$ different candidate mappings, where B_n is the n-th number in the fast-growing sequence of *Bell numbers*. But it is not necessary to follow this 'brute-force' approach, and two simple observations should in practice be of great help. More precisely, consider a step transition system $\mathfrak{ts} \overset{\mathrm{df}}{=} (S, T, s_{init})$. If it is generated by an ENCL-system with the locality mapping $\mathfrak{L}$, then the following hold, for every state $s \in S$:

– If $s \xrightarrow{u \uplus w}$ and $s \xrightarrow{u}$ then $\mathfrak{L}(e) \neq \mathfrak{L}(f)$, for all $e \in u$ and $f \in w$.
– If $s \xrightarrow{u}$ and there is no $w \subset u$ such that $s \xrightarrow{w}$ then $\mathfrak{L}(e) = \mathfrak{L}(f)$, for all $e, f \in u$.

Thus, for the example transition systems in Figures 2(a) and 4(c), we have respectively $\mathfrak{L}(e) = \mathfrak{L}(f)$ and $\mathfrak{L}(e) \neq \mathfrak{L}(f)$, and so the choice of localities we made was actually the only one which would work in these cases. On the other hand, for the step transition systems in Figures 4(a) and 6(a), the above rules do not provide any useful information. Indeed, in both cases we may take $\mathfrak{L}(e) = \mathfrak{L}(f)$ or $\mathfrak{L}(e) \neq \mathfrak{L}(f)$, and in each case synthesise a suitable ENCL-system, as shown in Figure 6(b) for the example in Figure 6(a). Note that these rules can be used for a quick decision that a step transition system is not a valid ENCL-transition

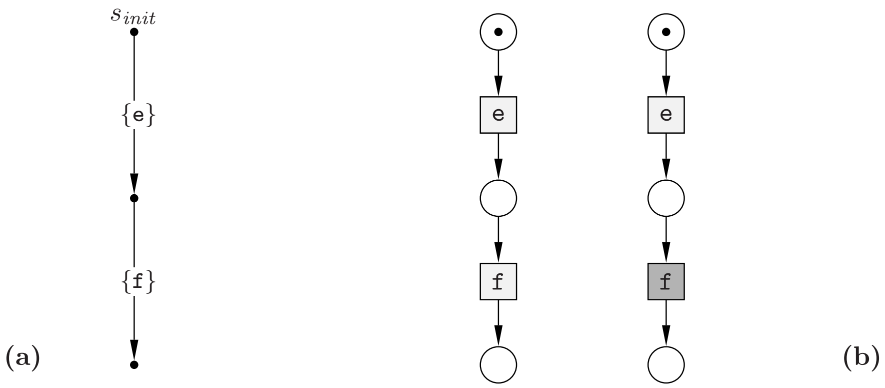


Fig. 6. A step transition system where no assumption about co-locating the events has been made **(a)**, and two corresponding ENCL-systems with different locality mappings **(b)**

system; for example, if we have $s_1 \xrightarrow{\{e,f\}}$ and $s_1 \xrightarrow{\{e\}}$ and $s_2 \xrightarrow{\{e,f\}}$ and $\neg s_2 \xrightarrow{\{e\}}$, for two distinct states, s_1 and s_2, of the same step transition system.

Previous work which appears to be closest to what has been proposed in this paper is due to Badouel and Darondeau [16]. It discusses the notion of a step transition system (generalising that introduced by Mukund [12]), which provides a much more general framework than the basic EN-transition systems; in particular, by dropping the assumption that a transition system should exhibit the so-called *intermediate state property*:

$$s \xrightarrow{\alpha+\beta} s' \implies \exists s'' : s \xrightarrow{\alpha} s'' \xrightarrow{\beta} s' \ .$$

But the step transition systems of [16] still exhibit a *subset property*:

$$s \xrightarrow{\alpha+\beta} \implies s \xrightarrow{\alpha} \ .$$

Neither of these properties holds for ENL-transition systems (and hence also for ENCL-transition systems). Instead, transition systems with localities enjoy their *weaker* version. More precisely, for ENL-transition systems we have:

$$s \xrightarrow{\alpha+\beta} s' \implies (s \xrightarrow{\alpha} s'' \implies s'' \xrightarrow{\beta} s' \wedge s \xrightarrow{\beta}) \ ,$$

and for ENCL-transition systems, we have:

$$s \xrightarrow{\alpha+\beta} \implies (s \xrightarrow{\alpha} \implies s \xrightarrow{\beta}) \ .$$

For example, the first of these properties implies that the transition system in Figure 4(c) cannot be generated by an ENL-system, and so the use of some context arcs is unavoidable as shown, e.g., in Figure 4(d). We feel that both properties might be useful in finding out whether (or to what extent) the theory of [16] could be adopted to work for the ENCL-transition systems as well.

Acknowledgment. This research was supported by the Epsrc project Casino, and Nsfc Grants 60373103 and 60433010.

References

1. Dasgupta, S., Potop-Butucaru, D., Caillaud, B., Yakovlev, A.: Moving from weakly endochronous systems to delay-insensitive circuits. Electr. Notes Theor. Comput. Sci. 146(2), 81–103 (2006)
2. Paun, G., Rozenberg, G.: A guide to membrane computing. Theor. Comput. Sci. 287(1), 73–100 (2002)
3. 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 (2005)
4. Koutny, M., Pietkiewicz-Koutny, M.: Transition systems of elementary net systems with localities. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 173–187. Springer, Heidelberg (2006)
5. Montanari, U., Rossi, F.: Contextual nets. Acta Inf. 32(6), 545–596 (1995)
6. Billington, J.: Protocol specification using p-graphs, a technique based on coloured petri nets, pp. 293–330 [22] DBLP:conf/ac/1996petri2
7. Donatelli, S., Franceschinis, G.: Modelling and analysis of distributed software using gspns, pp. 438–476 [22] DBLP:conf/ac/1996petri2
8. Esparza, J., Bruns, G.: Trapping mutual exclusion in the box calculus. Theor. Comput. Sci. 153(1-2), 95–128 (1996)
9. Kleijn, J., Koutny, M.: Synchrony and asynchrony in membrane systems. In: Workshop on Membrane Computing, pp. 20–39 (2006)
10. Ehrenfeucht, A., Rozenberg, G.: Theory of 2-structures, part i: Clans, basic subclasses, and morphisms. Theor. Comput. Sci. 70(3), 277–303 (1990)
11. Bernardinello, L., Michelis, G.D., Petruni, K., Vigna, S.: On the synchronic structure of transition systems. In: Desel, J. (ed.) Structures in Concurrency Theory, pp. 69–84. Springer, Heidelberg (1995)
12. Mukund, M.: Petri nets and step transition systems. Int. J. Found. Comput. Sci. 3(4), 443–478 (1992)
13. Nielsen, M., Winskel, G.: Models for concurrency. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.): Handbook of Logic in Computer Science, vol. 4, pp. 1–148 (1995)
14. Busi, N., Pinna, G.M.: Synthesis of nets with inhibitor arcs. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 151–165. Springer, Heidelberg (1997)
15. Pietkiewicz-Koutny, M.: The synthesis problem for elementary net systems with inhibitor arcs. Fundam. Inform. 40(2-3), 251–283 (1999)
16. Badouel, E., Darondeau, P.: Theory of regions. In: Reisig, W., Rozenberg, G. (eds.) Petri Nets. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1996)
17. Vogler, W.: Partial order semantics and read arcs. In: Privara, I., Ružička, P. (eds.) MFCS 1997. LNCS, vol. 1295, pp. 508–517. Springer, Heidelberg (1997)
18. Arnold, A.: Finite Transition Systems. Prentice-Hall International, Englewood Cliffs (1994)
19. Keller, R.M.: Formal verification of parallel programs. Commun. ACM 19(7), 371–384 (1976)

20. Nielsen, M., Rozenberg, G., Thiagarajan, P.S.: Elementary transition systems. Theor. Comput. Sci. 96(1), 3–33 (1992)
21. Pietkiewicz-Koutny, M.: Synthesising elementary net systems with inhibitor arcs from step transition systems. Fundam. Inform. 50(2), 175–203 (2002)
22. Reisig, W., Rozenberg, G. (eds.): Lectures on Petri Nets II: Applications. Applications, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996, LNCS, vol. 1492. Springer, Heidelberg (1998)

Appendix: Proof of Theorem 1

Clearly, $\mathfrak{ts}_{\mathfrak{encl}}$ is a step transition system. We need to prove that it satisfies the five axioms in Definition 5. Before doing this, we will show that, for every $b \in B$,

$$\mathfrak{r}_b \overset{\text{df}}{=} (\{c \in C_{\mathfrak{encl}} \mid b \in c\}, {}^\bullet b, b^\bullet, {}^\blacklozenge b, {}^\blacktriangleleft b)$$

is a (possibly trivial) c-region of $\mathfrak{ts}_{\mathfrak{encl}}$. Moreover, if $\varnothing \neq \|\mathfrak{r}_b\| \neq C_{\mathfrak{encl}}$ then $\mathfrak{r}_b$ is non-trivial.

To show that Definition 4 holds for $\mathfrak{r}_b$, we assume that $c \xrightarrow{u}_{\mathfrak{encl}} c'$ in $\mathfrak{ts}_{\mathfrak{encl}}$, and proceed as follows:

Proof of Definition 4(1) for $\mathfrak{r}_b$. We need to show that $c \in \|\mathfrak{r}_b\|$ and $c' \notin \|\mathfrak{r}_b\|$ implies $|u \cap {}^\bullet b| = 0$ and $|u \cap b^\bullet| = 1$.

From $c \in \|\mathfrak{r}_b\|$ ($c' \notin \|\mathfrak{r}_b\|$) it follows that $b \in c$ (resp. $b \notin c'$). Hence $b \in c \setminus c'$. From Proposition 1 we have $c \setminus c' = {}^\bullet u$ and $c' \setminus c = u^\bullet$. Hence $b \in {}^\bullet u$ and, as a consequence, there exists $e \in u$ such that $b \in {}^\bullet e$, and so $e \in b^\bullet$. We therefore have $e \in u \cap b^\bullet$. Hence $|u \cap b^\bullet| \geq 1$. Suppose that there is $f \neq e$ such that $f \in u \cap b^\bullet$. Then we have $f \in u$ and $b \in {}^\bullet f$ which implies $b \in {}^\bullet f \cap {}^\bullet e$, producing a contradiction with $e, f \in u \in \mathbb{U}_{\mathfrak{encl}}$. Hence $|u \cap b^\bullet| = 1$.

From $b \notin c'$ and $c' \setminus c = u^\bullet$, we have $b \notin u^\bullet$. Let $g \in u$ ($u \neq \varnothing$ by definition). Then $b \notin g^\bullet$, and so $g \notin {}^\bullet b$. Hence $|u \cap {}^\bullet b| = 0$.

Proof of Definition 4(2) for $\mathfrak{r}_b$. Can be proved similarly as Definition 4(1).

Proof of Definition 4(3) for $\mathfrak{r}_b$. We need to show that $u \cap {}^\blacklozenge b \neq \varnothing$ implies $c \notin \|\mathfrak{r}_b\|$. From $u \cap {}^\blacklozenge b \neq \varnothing$ we have that there is $e \in u$ such that $e \in {}^\blacklozenge b$ and so $b \in {}^\blacklozenge e$. Thus, since u is enabled at c in $\mathfrak{encl}$, $b \notin c$. Hence $c \notin \|\mathfrak{r}_b\|$.

Proof of Definition 4(4) for $\mathfrak{r}_b$. Can be proved similarly as Definition 4(3).

Proof of Definition 4(5) for $\mathfrak{r}_b$. We need to show that $u \cap b^\bullet \neq \varnothing$ implies $c \in \|\mathfrak{r}_b\|$ and $c' \notin \|\mathfrak{r}_b\|$. From Proposition 1 we have $c \setminus c' = {}^\bullet u$ and $c' \setminus c = u^\bullet$. From $u \cap b^\bullet \neq \varnothing$ we have that there is $e \in u$ such that $e \in b^\bullet$, and so $b \in {}^\bullet e$. Hence $b \in {}^\bullet u = c \setminus c'$, and so $b \in c$ and $b \notin c'$. Hence $c \in \|\mathfrak{r}_b\|$ and $c' \notin \|\mathfrak{r}_b\|$.

Proof of Definition 4(6) for $\mathfrak{r}_b$. Can be proved similarly as Definition 4(5).

Proof of Definition 4(7) for $\mathfrak{r}_b$. We need to show that ${}^\bullet b \cap {}^\blacklozenge b = b^\bullet \cap {}^\blacktriangleleft b = \varnothing$. This follows directly from the fact that $\mathfrak{encl}$ is an ENCL-system, where for every event e, the sets ${}^\bullet e$, $e^\bullet$, ${}^\blacklozenge e$ and ${}^\blacktriangleleft e$ are mutually disjoint.

Clearly, if $\varnothing \neq \|\mathfrak{r}_b\| \neq C_{\mathfrak{encl}}$ then $\mathfrak{r}_b$ is a non-trivial c-region.

Proof of AXIOM1. Follows directly from the definition of $C_{\mathfrak{encl}}$.

Proof of AXIOM2. We observe that if $e \in \mathcal{E}_{\mathfrak{ts}_{\mathfrak{encl}}}$ then $\{\mathfrak{r}_b \mid b \in {}^\bullet e\} \subseteq {}^\circ e$ and $\{\mathfrak{r}_b \mid b \in e^\bullet\} \subseteq e^\circ$ (follows from ${}^\bullet e \neq \varnothing \neq e^\bullet$, Proposition 2 and the definitions of ${}^\circ e$, e° and $\mathfrak{r}_b$). This and ${}^\bullet e \neq \varnothing \neq e^\bullet$ yields ${}^\circ e \neq \varnothing \neq e^\circ$.

Proof of AXIOM3. Suppose that $c \neq c'$ are two cases in $C_{\mathfrak{encl}}$. Without loss of generality, we may assume that there is $b \in c \setminus c'$. Hence $c \in \|\mathfrak{r}_b\|$ and $c' \notin \|\mathfrak{r}_b\|$. Thus, by the fact that $\mathfrak{r}_b$ is not trivial ($\varnothing \neq \|\mathfrak{r}_b\| \neq C_{\mathfrak{encl}}$) and $\mathfrak{r}_b \in \text{REG}_c \setminus \text{REG}_{c'}$, AXIOM3 holds.

Proof of AXIOM4. Suppose that $c \in C_{\mathfrak{encl}}$ and $u \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$ are such that ${}^\circ u^\triangleleft \subseteq \text{REG}_c$ and ${}^\lozenge u^\circ \cap \text{REG}_c = \varnothing$ and and there is no $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$ satisfying: $\mathfrak{L}(e) \in \mathfrak{L}(u)$ and ${}^\circ e^\triangleleft \subseteq \text{REG}_c$ and ${}^\lozenge e^\circ \cap \text{REG}_c = \varnothing$. We need to show that $c \xrightarrow{u}_{\mathfrak{encl}}$.

We have already shown that for $e \in \mathcal{E}_{\mathfrak{ts}_{\mathfrak{encl}}}$, $b \in {}^\bullet e$ implies $\mathfrak{r}_b \in {}^\circ e$, and $b \in e^\bullet$ implies $\mathfrak{r}_b \in e^\circ$. From this and $u \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$ we have that $u \in \mathbb{U}_{\mathfrak{encl}}$.
First we show ${}^\bullet u \subseteq c$. Let $e \in u$. Consider $b \in {}^\bullet e$. We have already shown that this implies $\mathfrak{r}_b \in {}^\circ e$. From ${}^\circ u \subseteq \text{REG}_c$, we have that $\mathfrak{r}_b \in \text{REG}_c$, and so $c \in \|\mathfrak{r}_b\|$. Consequently, $b \in c$. Hence, for all $e \in u$ we have ${}^\bullet e \subseteq c$, and so ${}^\bullet u \subseteq c$.

Now we show that $u^\bullet \cap c = \varnothing$. Let $e \in u$. Consider $b \in e^\bullet$. We have already shown that this implies $\mathfrak{r}_b \in e^\circ$. From $u^\circ \cap \text{REG}_c = \varnothing$, we have that $\mathfrak{r}_b \notin \text{REG}_c$, and so $c \notin \|\mathfrak{r}_b\|$. Consequently, $b \notin c$. Hence, for all $e \in u$ we have $e^\bullet \cap c = \varnothing$, and so $u^\bullet \cap c = \varnothing$.

Now we show that ${}^\blacklozenge u \cap c = \varnothing$. Suppose to the contrary that ${}^\blacklozenge u \cap c \neq \varnothing$. Then there is $e \in u$ such that ${}^\blacklozenge e \cap c \neq \varnothing$, and as a consequence there is $b \in {}^\blacklozenge e$ such that $b \in c$. Hence, $c \in \|\mathfrak{r}_b\|$ and so $\|\mathfrak{r}_b\| \neq \varnothing$. We now prove that $\|\mathfrak{r}_b\| \neq C_{\mathfrak{encl}}$. Suppose $\|\mathfrak{r}_b\| = \{c \in C_{\mathfrak{encl}} \mid b \in c\} = C_{\mathfrak{encl}}$. Then b is a condition present in every case c of $\mathfrak{encl}$ making it impossible for any step containing e to be enabled ($b \in {}^\blacklozenge e$). This, in turn, contradicts the fact that $e \in \mathcal{E}_{\mathfrak{ts}_{\mathfrak{encl}}}$ (as an event in $u \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$) and must appear in some step labelling a transition from $\mathfrak{ts}_{\mathfrak{encl}}$. Hence $\|\mathfrak{r}_b\| \neq C_{\mathfrak{encl}}$, and so $\mathfrak{r}_b$ is a non-trivial c-region. From $b \in {}^\blacklozenge e$ we have $e \in {}^\blacklozenge b = {}^\blacklozenge \mathfrak{r}_b$, which means that $\mathfrak{r}_b \in {}^\lozenge e$. Consequently, $\mathfrak{r}_b \in {}^\lozenge u$. From this and ${}^\lozenge u \cap \text{REG}_c = \varnothing$ we have $\mathfrak{r}_b \notin \text{REG}_c$, and so $c \notin \|\mathfrak{r}_b\|$. Consequently $b \notin c$, and so we obtained a contradiction. Hence ${}^\blacklozenge u \cap c = \varnothing$.

Now we show that ${}^\blacktriangleleft u \subseteq c$. Suppose to the contrary that there is $b \in {}^\blacktriangleleft u \setminus c$. From $b \in {}^\blacktriangleleft u$ we have that there is $e \in u$ such that $b \in {}^\blacktriangleleft e$. From $b \notin c$ we have that $c \notin \|\mathfrak{r}_b\|$, and so $\|\mathfrak{r}_b\| \neq C_{\mathfrak{encl}}$. We now prove that $\|\mathfrak{r}_b\| \neq \varnothing$. Assume that $\|\mathfrak{r}_b\| = \varnothing$. This implies that, for all $c \in C_{\mathfrak{encl}}$, $b \notin c$. But this would make it impossible to execute any step containing e in $\mathfrak{encl}$. This, in turn, contradicts the fact that $e \in \mathcal{E}_{\mathfrak{ts}_{\mathfrak{encl}}}$ and so it must appear in some step labelling a transition in

$\mathfrak{ts}_{\mathfrak{encl}}$. Hence $\|\mathfrak{r}_b\| \neq \varnothing$, and so the c-region $\mathfrak{r}_b$ is non-trivial. From $b \in {}^{\blacktriangleleft}e$ we have that $e \in {}^{\blacktriangleleft}b = {}^{\blacktriangleleft}\mathfrak{r}_b$. Consequently, we have that $\mathfrak{r}_b \in {}^{\triangleleft}e$, and so $\mathfrak{r}_b \in {}^{\triangleleft}u$. From this and ${}^{\triangleleft}u \subseteq \textsc{Reg}_c$ we have that $\mathfrak{r}_b \in \textsc{Reg}_c$, and so $c \in \|\mathfrak{r}_b\|$. Consequently $b \in c$, and so we obtained a contradiction. Hence ${}^{\blacktriangleleft}u \subseteq c$.

All what remains to be shown is that there is no step $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{encl}}$ satisfying: $\mathfrak{L}(e) \in \mathfrak{L}(u)$, ${}^{\bullet}e^{\blacktriangleleft} \subseteq c$ and ${}^{\blacklozenge}e^{\bullet} \cap c = \varnothing$. Suppose that this is not the case, and $u \uplus \{e_1\} \in \mathbb{U}_{\mathfrak{encl}}$ is a step satisfying these conditions. We consider two cases.

Case 1: There is no $u \uplus \{e_1\} \uplus \{f\} \in \mathbb{U}_{\mathfrak{encl}}$ such that $\mathfrak{L}(f) \in \mathfrak{L}(u \uplus \{e_1\})$, ${}^{\bullet}f^{\blacktriangleleft} \subseteq c$ and ${}^{\blacklozenge}f^{\bullet} \cap c = \varnothing$. This implies $c \xrightarrow{u \uplus \{e_1\}}_{\mathfrak{encl}}$. By Proposition 12, we have that $u \uplus \{e_1\} \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$. Moreover, $\mathfrak{L}(e_1) \in \mathfrak{L}(u)$ and, by Propositions 7(3) and 10, we have ${}^{\circ}(u \uplus \{e_1\})^{\triangleleft} \subseteq \textsc{Reg}_c$ and ${}^{\Diamond}(u \uplus \{e_1\})^{\circ} \cap \textsc{Reg}_c = \varnothing$. We therefore obtained a contradiction with our assumptions.
Case 2: We can find $u \uplus \{e_1\} \uplus \{e_2\} \in \mathbb{U}_{\mathfrak{encl}}$ such that $\mathfrak{L}(e_2) \in \mathfrak{L}(u \uplus \{e_1\})$, ${}^{\bullet}e_2{}^{\blacktriangleleft} \subseteq c$ and ${}^{\blacklozenge}e_2{}^{\bullet} \cap c = \varnothing$. Then we consider Cases 1 and 2 again, taking $u \uplus \{e_1\} \uplus \{e_2\}$ instead of $u \uplus \{e_1\}$. Since the number of events in E is finite, we will eventually end up in Case 1. This means that, eventually, we will obtain a contradiction.

Proof of Axiom5. We need to show that if $c \xrightarrow{u}_{\mathfrak{encl}}$ then there is no $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$ satisfying $\mathfrak{L}(e) \in \mathfrak{L}(u)$, ${}^{\circ}e^{\triangleleft} \subseteq \textsc{Reg}_c$ and ${}^{\Diamond}e^{\circ} \cap \textsc{Reg}_c = \varnothing$.
Suppose to the contrary that there is $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$ as above (†).

We have already shown that for $e \in \mathcal{E}_{\mathfrak{ts}_{\mathfrak{encl}}}$, $b \in {}^{\bullet}e$ implies $\mathfrak{r}_b \in {}^{\circ}e$, and $b \in e^{\bullet}$ implies $\mathfrak{r}_b \in e^{\circ}$. From this and $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{ts}_{\mathfrak{encl}}}$ we have $u \uplus \{e\} \in \mathbb{U}_{\mathfrak{encl}}$.

We will show that ${}^{\bullet}e \subseteq c$. Consider $b \in {}^{\bullet}e$. We have that $b \in {}^{\bullet}e$ implies $\mathfrak{r}_b \in {}^{\circ}e$. But ${}^{\circ}e \subseteq \textsc{Reg}_c$, and so $\mathfrak{r}_b \in \textsc{Reg}_c$. This means that $c \in \|\mathfrak{r}_b\|$, and consequently $b \in c$. Hence ${}^{\bullet}e \subseteq c$.

We now show that $e^{\bullet} \cap c = \varnothing$. Consider $b \in e^{\bullet}$. We have that $b \in e^{\bullet}$ implies $\mathfrak{r}_b \in e^{\circ}$. But $e^{\circ} \cap \textsc{Reg}_c = \varnothing$, and so $\mathfrak{r}_b \notin \textsc{Reg}_c$. This means that $c \notin \|\mathfrak{r}_b\|$, and consequently, $b \notin c$. Hence $e^{\bullet} \cap c = \varnothing$.

Now we show that ${}^{\blacklozenge}e \cap c = \varnothing$. Suppose to the contrary that $b \in {}^{\blacklozenge}e \cap c \neq \varnothing$. We have already shown in the proof of Axiom4 that for $e \in \mathcal{E}_{\mathfrak{ts}_{\mathfrak{encl}}}$, $b \in {}^{\blacklozenge}e \cap c$ implies $\mathfrak{r}_b \in {}^{\Diamond}e$. But ${}^{\Diamond}e \cap \textsc{Reg}_c = \varnothing$, so $\mathfrak{r}_b \notin \textsc{Reg}_c$. This means $c \notin \|\mathfrak{r}_b\|$, and so $b \notin c$, a contradiction.

Finally, we show that ${}^{\blacktriangleleft}e \subseteq c$. Suppose to the contrary that there is $b \in {}^{\blacktriangleleft}e \setminus c$. We have already shown in the proof of Axiom4 that for $e \in \mathcal{E}_{\mathfrak{ts}_{\mathfrak{encl}}}$, $b \in {}^{\blacktriangleleft}e \setminus c$ implies $\mathfrak{r}_b \in {}^{\triangleleft}e$. But ${}^{\triangleleft}e \subseteq \textsc{Reg}_c$, so $\mathfrak{r}_b \in \textsc{Reg}_c$. This means that $c \in \|\mathfrak{r}_b\|$ and, consequently $b \in c$, a contradiction.

As a result, assuming that (†) holds leads to a contradiction with $c \xrightarrow{u}_{\mathfrak{encl}}$.

Nets with Tokens Which Carry Data

Ranko Lazić[1,⋆], Tom Newcomb[2], Joël Ouaknine[2],
A.W. Roscoe[2], and James Worrell[2]

[1] Department of Computer Science, University of Warwick, UK
[2] Computing Laboratory, University of Oxford, UK

Abstract. We study data nets, a generalisation of Petri nets in which tokens carry data from linearly-ordered infinite domains and in which whole-place operations such as resets and transfers are possible. Data nets subsume several known classes of infinite-state systems, including multiset rewriting systems and polymorphic systems with arrays.

We show that coverability and termination are decidable for arbitrary data nets, and that boundedness is decidable for data nets in which whole-place operations are restricted to transfers. By providing an encoding of lossy channel systems into data nets without whole-place operations, we establish that coverability, termination and boundedness for the latter class have non-primitive recursive complexity. The main result of the paper is that, even for unordered data domains (i.e., with only the equality predicate), each of the three verification problems for data nets without whole-place operations has non-elementary complexity.

1 Introduction

Petri nets (e.g., [1]) are a fundamental model of concurrent systems. Being more expressive than finite-state machines and less than Turing-powerful, Petri nets have an established wide range of applications and a variety of analysis tools (e.g., [2]).

The analysis tools are based on the extensive literature on decidability and complexity of verification problems ([3] is a comprehensive survey). In this paper, we focus on three basic decision problems, to which a number of other verification questions can be reduced:

Coverability: Is a marking reachable which is greater than or equal to a given marking?
Termination: Are all computations finite?
Boundedness: Is the set of all reachable markings finite?

By the results in [4,5], each of coverability, termination and boundedness is ExpSpace-complete for Petri nets.

Many extensions of Petri nets preserve decidability of various verification problems. Notably, affine well-structured nets were formulated in [6] as an elegant extension of Petri nets by whole-place operations. The latter are resets,

⋆ Supported by the EPSRC (GR/S52759/01) and the Intel Corporation.

which empty a place, and transfers, which take all tokens from a place and put them onto one or more specified places (possibly several times). Hence, two subclasses of affine WSNs are reset nets and transfer nets, in which whole-place operations are restricted to resets and to transfers, respectively. As shown in [6], coverability and termination for affine WSNs, and boundedness for transfer nets, are decidable. However, compared with Petri nets, there is a dramatic increase in complexity: it follows from the results on lossy channel systems in [7] that coverability and termination for reset nets and transfer nets, and boundedness for transfer nets, are not primitive recursive.[1] It was proved in [9] that boundedness for reset nets is undecidable.

Another important direction of extending Petri nets is by allowing tokens to carry data from infinite domains. (Data from finite domains do not increase expressiveness.) For example, in timed Petri nets [10], each token is equipped with a real-valued clock which represents the age of the token. Multiset rewriting specifications over constraint systems $\mathcal{C}$ [11,12] can be seen as extensions of Petri nets in which tokens may carry data from the domain of $\mathcal{C}$ and transitions can be constrained using $\mathcal{C}$. In mobile synchronizing Petri nets [13], tokens may carry identifiers from an infinite domain, and transitions may require that an identifier be fresh (i.e., not currently carried by any token).

In this paper, we focus on the following two questions:

(1) Is there a general extension of Petri nets in which tokens carry data from infinite domains, in which whole-place operations are possible, and such that coverability, termination and boundedness are decidable (either for the whole class of extended nets or for interesting subclasses)?

(2) If the answer to the previous question is positive, and if we restrict to the subclass without whole-place operations, do coverability, termination and boundedness remain ExpSpace-complete (as for Petri nets), or are their complexities greater? What happens if we restrict further to the simplest data domains, i.e. those with only the equality predicate?

Data nets. To answer question (1), we define data nets, in which tokens carry data from linearly-ordered infinite domains. As in Petri nets, transitions consume and produce tokens. For a transition to be firable, we can require that the data which are carried by the tokens to be consumed are ordered in a certain way. In addition to such data, transitions can choose finitely many other data, which satisfy further ordering constraints and which may or may not be present in the current marking. In the production phase, tokens which carry either kind of data can be put into the marking. Data nets also support whole-place operations.

In the next few paragraphs, we introduce data nets in an informal but detailed manner, for clarity of the subsequent discussion of contributions of the paper and relations to the literature. As an alternative order of presentation, the reader may wish to postpone the following and read it in conjunction with Section 2.2, where data nets are defined formally.

[1] Recall the Ritchie-Cobham property [8, page 297]: a decision problem (i.e. a set) is primitive recursive iff it is solvable in primitive recursive time/space.

Data nets are based on affine WSNs [6]. Markings of an affine WSN are vectors in $\mathbb{N}^P$, where P is the finite set of all places. A transition t of an affine WSN is given by vectors $F_t, H_t \in \mathbb{N}^P$ and a square matrix $G_t \in \mathbb{N}^{P \times P}$. Such a transition is firable from a marking m iff $m \geq F_t$, and in that case it produces the marking $(m - F_t)G_t + H_t$. Whole-place operations are performed by the multiplication with G_t.

Since a linear ordering $\preceq$ is the only operation available on data, markings of data nets are finite sequences of vectors in $\mathbb{N}^P \setminus \{\mathbf{0}\}$. Each index j of such a marking s corresponds to an implicit datum d_j, and we have that $j \leq j'$ iff $d_j \preceq d_{j'}$. For each $p \in P$, $s(j)(p)$ is the number of tokens which carry d_j and are at place p. We say that such tokens are at index j. Now, each transition t has an arity $\alpha_t \in \mathbb{N}$. For a transition t to be fired from a marking s, we choose nondeterministically α_t mutually distinct data. Some of those data may be fresh (i.e., not carried by any token in s), so picking the α_t data is formalised by first expanding s to a finite sequence $s_\dagger$ by inserting the vector $\mathbf{0}$ at arbitrary positions, and then picking an increasing (in particular, injective) mapping

$$\iota : \{1, \ldots, \alpha_t\} \to \{1, \ldots, |s_\dagger|\}$$

such that each occurrence of $\mathbf{0}$ is in its range. Now, such a mapping ι partitions $\{1, \ldots, |s_\dagger|\}$ into α_t singletons and $\alpha_t + 1$ contiguous "regions" as follows, where the $\mathit{Reg}_{(i,i+1)}$ are region identifiers:

$$\underbrace{1, \ldots, \iota(1) - 1}_{\mathit{Reg}_{(0,1)}}, \iota(1), \underbrace{\iota(1) + 1, \ldots, \iota(2) - 1}_{\mathit{Reg}_{(1,2)}}, \ldots, \iota(\alpha_t), \underbrace{\iota(\alpha_t) + 1, \ldots, |s_\dagger|}_{\mathit{Reg}_{(\alpha_t,\alpha_t+1)}}$$

The action of t on s with respect to $s_\dagger$ and ι is determined by vectors F_t and H_t, and a square matrix G_t, whose elements are natural numbers, and which are indexed by

$$(\{1, \ldots, \alpha_t\} \cup \{\mathit{Reg}_{(i,i+1)} : 0 \leq i \leq \alpha_t\}) \times P$$

It consists of the following stages, where $i, i' \in \{1, \ldots, \alpha_t\}$, $R, R' \in \{\mathit{Reg}_{(i,i+1)} : 0 \leq i \leq \alpha_t\}$ and $p, p' \in P$.

subtraction: for each i and p, $F_t(i, p)$ tokens at index $\iota(i)$ are taken from p;[2]
multiplication: all tokens are taken simultaneously, and then:
- for each token taken from p at index $\iota(i)$, $G_t(i, p, i', p')$ tokens are put onto p' at index $\iota(i')$, and for each j' in region R', $G_t(i, p, R', p')$ tokens are put onto p' at index j';
- for each token taken from p at index j in region R, $G_t(R, p, i', p')$ tokens are put onto p' at index $\iota(i')$, and $G_t(R, p, R, p')$ tokens are put onto p' at index j;

addition: for each i and p, $H_t(i, p)$ tokens are put onto p at index $\iota(i)$, and for each j in region R and p, $H_t(R, p)$ tokens are put onto p at index j.

[2] In order to have well-structuredness (see Proposition 7) and for simplicity, entries $F_t(R, p)$ are not used, and neither are entries $G_t(R, p, R', p')$ with $R \neq R'$, so they are assumed to be 0.

Example 1. Consider $P = \{p_1, p_2\}$ and a transition t with $\alpha_t = 1$ given by:

$$\begin{array}{c|ccc} F_t & Reg_{(0,1)} & 1 & Reg_{(1,2)} \\ \hline & 0\;0 & 1\;1 & 0\;0 \\ \hline & p_1\;p_2 & p_1\;p_2 & p_1\;p_2 \end{array}$$

$$\begin{array}{c|ccc} H_t & Reg_{(0,1)} & 1 & Reg_{(1,2)} \\ \hline & 0\;0 & 2\;1 & 6\;0 \\ \hline & p_1\;p_2 & p_1\;p_2 & p_1\;p_2 \end{array}$$

$$\begin{array}{c|ccc|c} G_t & Reg_{(0,1)} & 1 & Reg_{(1,2)} & \\ \hline \multirow{2}{*}{$Reg_{(0,1)}$} & 0\;1 & 0\;0 & 0\;0 & p_1 \\ & 1\;0 & 0\;0 & 0\;0 & p_2 \\ \multirow{2}{*}{1} & 0\;0 & 2\;0 & 3\;0 & p_1 \\ & 0\;0 & 0\;1 & 3\;0 & p_2 \\ \multirow{2}{*}{$Reg_{(1,2)}$} & 0\;0 & 0\;0 & 1\;0 & p_1 \\ & 0\;0 & 0\;2 & 0\;1 & p_2 \\ \hline & p_1\;p_2 & p_1\;p_2 & p_1\;p_2 & \end{array}$$

From a marking s, in terms of data represented by the indices of s, transition t is fired as follows:

1. a datum d is chosen nondeterministically, such that each of p_1 and p_2 contain at least 1 token carrying d (so, d cannot be fresh);
2. for each datum $d' \prec d$, all tokens at p_1 carrying d' are transferred to p_2, and vice-versa;
3. for each token at p_1 or p_2 carrying d, and each $d' \succ d$, 3 tokens carrying d' are put onto p_1;
4. the number of tokens at p_1 carrying d is multiplied by 2;
5. for each token at p_2 carrying $d' \succ d$, 2 tokens carrying d are put onto p_2.

Since $H_t = F_t G_t$, the addition stage of performing t exactly "undoes" the subtraction stage, so t performs only whole-place operations.

In Section 2.2, the above will be formalised so that t is firable from s with respect to $s_\dagger$ and ι iff $s_\dagger \geq [\![F_t]\!]_\iota^{|s_\dagger|}$, and in that case it produces the marking obtained from $(s_\dagger - [\![F_t]\!]_\iota^{|s_\dagger|})[\![G_t]\!]_\iota^{|s_\dagger|} + [\![H_t]\!]_\iota^{|s_\dagger|}$ by removing each entry $\mathbf{0}$, where $[\![F_t]\!]_\iota^{|s_\dagger|}$, $[\![G_t]\!]_\iota^{|s_\dagger|}$ and $[\![H_t]\!]_\iota^{|s_\dagger|}$ are appropriate "expansions" of F_t, G_t and H_t, indexed by $\{1, \ldots, |s_\dagger|\} \times P$.

Since vectors $\mathbf{0}$ which correspond to fresh data can be inserted at arbitrary positions to fire a transition, the linear ordering on data is assumed to be dense and without least and greatest elements. Having a least or greatest element can easily be simulated, and density is not a restriction when considering only finite computations (as is the case for the coverability problem).

We show that affine WSNs [6] are equivalent to a class of data nets whose transitions have arity 1. Data nets also subsume timed Petri nets [10] and timed networks [14], in the sense that systems obtained after quotienting by time regions can be simulated by data nets, where the data domain is fractional parts of clock values. Monadic multiset rewriting specifications over order constraints on rationals or reals [11] and over gap-order constrains on integers [12] can be translated to data nets, subject to the remarks above about density. Mobile synchronizing petri nets [13], lossy channel systems [15], and polymorphic systems with one array of type $\langle X, \leq \rangle \to \{1, \ldots, n\}$ or with two arrays of types $\langle X, = \rangle \to \langle Y, \leq \rangle$ and $\langle X, = \rangle \to \{1, \ldots, n\}$ [16,17], can also be expressed using data nets.

Decidability. Using the theory of well-structured transition systems [18], we prove that coverability and termination for arbitrary data nets, and boundedness for data nets in which whole-place operations are restricted to transfers, are decidable. Thus, question (1) posed above is answered positively. The decidability of coverability for data nets subsumes the results in [6,10,14,11,12,13,15,16,17] that coverability is decidable for the respective classes of infinite-state systems mentioned above, and in most cases the proof in this paper is more succinct.

Hardness. To question (2) above, we obtain the following answers. We say that a data net is *Petri* iff it does not contain whole-place operations, and *unordered* iff it makes use only of equality between data (and not of the linear ordering).

– By providing a translation from lossy channel systems to Petri data nets, we establish that coverability, termination and boundedness for the latter class are not primitive recursive. The encoding uses the linear ordering on the data domain, for picking fresh data which are employed in simulating writes to channels.
– The main result of the paper is that coverability, termination and boundedness for unordered Petri data nets are not elementary, i.e., their computational complexities cannot be bounded by towers of exponentials of fixed heights. That is a surprising result, since unordered Petri data nets are highly constrained systems. In particular, they do not provide a mechanism for ensuring that a datum chosen in a transition is fresh (i.e., not present in the current marking). The result is proved by simulating a hierarchy of bounded counters, which is reminiscent of the "rulers" construction of Meyer and Stockmeyer (e.g., [19]).

By translating Petri data nets and unordered Petri data nets to subclasses of systems in [11,12,13,16,17], the two hardness results yield the same lower bounds for corresponding decision problems for such subclasses. In particular, we obtain non-elementariness of verifying monadic multiset rewriting specifications with only equality constraints [11] and of verifying polymorphic systems with two arrays of types $\langle X, =\rangle \to \langle Y, =\rangle$ and $\langle X, =\rangle \to \{1, \ldots, n\}$ [16].

Paper organisation. Section 2 contains preliminaries, including definitions of data nets and of several relevant subclasses, some basic results, and an example. In Section 3, we present the translation from lossy channel systems to Petri data nets. Sections 4 and 5 contain the decidability and hardness results. Some remaining open problems are discussed in Section 6.

2 Preliminaries

Sets, quasi-orders and mappings. For $n \in \mathbb{N}$, let $[n] = \{1, \ldots, n\}$. We write $\mathbb{N}_\omega$ for $\mathbb{N} \cup \{\omega\}$. The linear ordering $\leq$ on $\mathbb{N}$ is extended to $\mathbb{N}_\omega$ by having $n < \omega$ for each $n \in \mathbb{N}$.

A set A and a relation $\preceq$ on A form a *quasi-order* iff $\preceq$ is reflexive and transitive. We write $a_1 \prec a_2$ iff $a_1 \preceq a_2$ and $a_2 \not\preceq a_1$.

For any $A' \subseteq A$, its upward closure is $\uparrow A' = \{a \in A : \exists a' \in A' \cdot a' \preceq a\}$. We say that A' is upwards-closed iff $A' = \uparrow A'$. A *basis* of an upwards-closed set A' is a subset A'' such that $A' = \uparrow A''$. Downward closure (written $\downarrow A'$), closedness and bases are defined symmetrically.

A mapping f from a quasi-order $\langle A, \preceq \rangle$ to a quasi-order $\langle A', \preceq' \rangle$ is *increasing* iff $a_1 \prec a_2 \Rightarrow f(a_1) \prec' f(a_2)$.

Vectors and matrices. For sets A and B, let A^B denote the set of all B-indexed vectors of elements of A, i.e., the set of all mappings $B \to A$. For example, $\mathbb{N}^{[n] \times [n']}$ is the set of all $n \times n'$ matrices of natural numbers. For $a \in A$, let $\mathbf{a} \in A^B$ denote the vector whose each entry equals a. Let $Id \in \mathbb{N}^{B \times B}$ denote the identity square matrix.

A quasi-ordering $\preceq$ on A induces the following quasi-ordering on A^B: $v \preceq v'$ iff $v(b) \preceq v'(b)$ for all $b \in B$.

Sequences and bags. For a set A, let $Seq(A)$ denote the set of all finite sequences of elements of A. For $s \in Seq(A)$, let $|s|$ denote the length of s, and $s(1), \ldots, s(|s|)$ denote its elements.

For $s, s' \in Seq(A)$ and $a \in A$, we say that s' is an *a-expansion* of s (equivalently, s is the *a-contraction* of s') iff s is obtained by removing each occurrence of a from s'.

For $s, s' \in Seq(A)$, we write $s \sim s'$ iff s' can be obtained from s by permuting its entries. We define the set $Bag(A)$ of all finite bags (i.e., multisets) of elements of A as the set of all equivalence classes of $\sim$. Let $\overline{s}$ denote the equivalence class of s, i.e., the bag with the same elements as s.

Suppose $\langle A, \preceq \rangle$ is a quasi-order. The quasi-ordering $\preceq$ induces quasi-orderings on $Seq(A)$ and $Bag(A)$ as follows. For $s, s' \in Seq(A)$, we write $s \preceq s'$ iff there exists an increasing $\iota : [|s|] \to [|s'|]$ such that $s(i) \preceq s'(\iota(i))$ for all $i \in [|s|]$. For $b, b' \in Bag(A)$, we write $b \preceq b'$ iff there exist $s \in b$ and $s' \in b'$ such that $s \preceq s'$.

Well-quasi-orderings. A quasi-ordering $\preceq$ on a set A is a well-quasi-ordering iff, for every infinite sequence $a_1, a_2, \ldots \in A$, there exist $i < j$ such that $a_i \preceq a_j$.

Proposition 2 ([20]). *Whenever $\preceq$ is a well-quasi-ordering on a set A, the induced orderings on $Seq(A)$ and $Bag(A)$ also are well-quasi-orderings.*

2.1 Affine Well-Structured Nets

We recall the notion of affine well-structured net [6].[3] Such a net is a tuple $\langle P, T, F, G, H \rangle$ such that P is a finite set of places, T is a finite set of transitions, and for each $t \in T$, F_t and H_t are vectors in $\mathbb{N}^P$, and G_t is a matrix in $\mathbb{N}^{P \times P}$.

Markings of an affine WSN $\langle P, T, F, G, H \rangle$ are vectors in $\mathbb{N}^P$. A marking m' can be obtained from a marking m by firing a transition $t \in T$, written $m \xrightarrow{t} m'$, iff $m \geq F_t$ and $m' = (m - F_t)G_t + H_t$.

[3] For technical reasons, the formalisation of affine WSNs in this paper is slightly different, but equivalent.

As was shown in [6], Petri nets and many of their known extensions are special cases of affine WSNs. In particular, Petri nets and their extensions by (generalised) resets and transfers are equivalent to the classes of affine WSNs $\langle P, T, F, G, H\rangle$ determined by the following restrictions:

Petri nets: $\forall t \in T \cdot G_t = Id$
reset nets: $\forall t \in T \cdot G_t \leq Id$
transfer nets: $\forall t \in T, p \in P \cdot \exists p' \in P \cdot G_t(p, p') > 0$

2.2 Data Nets

Given $n \in \mathbb{N}$, let $Regs(n) = \{Reg_{(i,i+1)} : 0 \leq i \leq n\}$. For each $0 \leq i \leq n$, $m \geq n$ and increasing $\iota : [n] \to [m]$, let $[\![Reg_{(i,i+1)}]\!]^m_\iota = \{j \in [m] : \iota(i) < j < \iota(i+1)\}$, where by convention $\iota(0) = 0$ and $\iota(n+1) = m+1$.

A *data net* is a tuple $\langle P, T, \alpha, F, G, H\rangle$ such that:

- P is a finite set of places;
- T is a finite set of transitions;
- for each $t \in T$, $\alpha_t \in \mathbb{N}$ specifies the arity of t;
- for each $t \in T$, $F_t \in \mathbb{N}^{([\alpha_t] \cup Regs(\alpha_t)) \times P}$, and $F_t(R, p) = 0$ whenever $R \in Regs(\alpha_t)$ and $p \in P$;
- for each $t \in T$, $G_t \in \mathbb{N}^{(([\alpha_t] \cup Regs(\alpha_t)) \times P)^2}$, and $G_t(R, p, R', p') = 0$ whenever $R, R' \in Regs(\alpha_t)$, $R \neq R'$ and $p, p' \in P$;
- for each $t \in T$, $H_t \in \mathbb{N}^{([\alpha_t] \cup Regs(\alpha_t)) \times P}$.

Suppose $\langle P, T, \alpha, F, G, H\rangle$ is a data net, and $t \in T$. Any $m \geq \alpha_t$ and increasing $\iota : [\alpha_t] \to [m]$ determine the following instances of F_t, G_t and H_t:

- $[\![F_t]\!]^m_\iota \in \mathbb{N}^{[m] \times P}$ is defined by

$$[\![F_t]\!]^m_\iota(\iota(i), p) = F_t(i, p) \qquad [\![F_t]\!]^m_\iota(j, p) = F_t(R, p) \text{ for } j \in [\![R]\!]^m_\iota$$

- $[\![G_t]\!]^m_\iota \in \mathbb{N}^{([m] \times P)^2}$ is defined by

$$\begin{aligned} [\![G_t]\!]^m_\iota(\iota(i), p, \iota(i'), p') &= G_t(i, p, i', p') & \\ [\![G_t]\!]^m_\iota(\iota(i), p, j', p') &= G_t(i, p, R, p') & \text{for } j' \in [\![R]\!]^m_\iota \\ [\![G_t]\!]^m_\iota(j, p, \iota(i'), p') &= G_t(R, p, i', p') & \text{for } j \in [\![R]\!]^m_\iota \\ [\![G_t]\!]^m_\iota(j, p, j, p') &= G_t(R, p, R, p') & \text{for } j \in [\![R]\!]^m_\iota \\ [\![G_t]\!]^m_\iota(j, p, j', p') &= 0 & \text{otherwise} \end{aligned}$$

- $[\![H_t]\!]^m_\iota \in \mathbb{N}^{[m] \times P}$ is defined in the same way as $[\![F_t]\!]^m_\iota$.

A *marking* of a data net $\langle P, T, \alpha, F, G, H\rangle$ is a finite sequence of vectors in $\mathbb{N}^P \setminus \{\mathbf{0}\}$. A marking s' can be obtained from a marking s by firing a transition

$t \in T$, written $s \xrightarrow{t} s'$, iff there exist a $\mathbf{0}$-expansion $s_\dagger$ of s and an increasing $\iota : [\alpha_t] \to [|s_\dagger|]$ such that:[4]

(i) $\{j : s_\dagger(j) = \mathbf{0}\} \subseteq \mathit{Range}(\iota)$;
(ii) $s_\dagger \geq [\![F_t]\!]_\iota^{|s_\dagger|}$;
(iii) s' is the $\mathbf{0}$-contraction of $(s_\dagger - [\![F_t]\!]_\iota^{|s_\dagger|})[\![G_t]\!]_\iota^{|s_\dagger|} + [\![H_t]\!]_\iota^{|s_\dagger|}$.

We may also write $s \xrightarrow{t, s_\dagger, \iota} s'$, or just $s \to s'$.

Proposition 3. *For any data net, its transition system* $\langle \mathit{Seq}(\mathbb{N}^P \setminus \{\mathbf{0}\}), \to \rangle$ *is finitely branching.*

2.3 Decision Problems

We consider the following standard problems:

Coverability: Given a data net, and markings s and s', to decide whether some marking $s'' \geq s'$ is reachable from s.
Termination: Given a data net, and a marking s, to decide whether all computations from s are finite.
Boundedness: Given a data net, and a marking s, to decide whether the set of all markings reachable from s is finite.

Coverability, termination and boundedness for affine WSNs are defined in the same way.

2.4 Classes of Data Nets

We now define several classes of data nets. Figure 1 shows the inclusions among classes of data nets and affine well-structured nets in Propositions 5, 6, 8 and 9 below. In addition, the mapping $\mathcal{N} \mapsto \widetilde{\mathcal{N}}$ and its inverse (see Proposition 6) provide a correspondence between unary transfer data nets (resp., unary Petri data nets) and transfer nets (resp., Petri nets). The dashed line represents the fact that Proposition 9 does not provide a reduction for the boundedness problem.

Unordered data nets. A data net $\langle P, T, \alpha, F, G, H \rangle$ is unordered iff:

(i) for each $t \in T$, $R, R' \in \mathit{Regs}(\alpha_t)$ and $p, p' \in P$, we have $G_t(R, p, R, p') = G_t(R', p, R', p')$ and $H_t(R, p) = H_t(R', p)$;
(ii) for each $t \in T$ and permutation π of $[\alpha_t]$, there exists $t' \in T$ such that $F_{t'}$, $G_{t'}$ and $H_{t'}$ are obtained from F_t, G_t and H_t (respectively) by applying π to each index in $[\alpha_t]$.

Given an unordered data net $\langle P, T, \alpha, F, G, H \rangle$, we write $t \sim t'$ iff t and t' have the property in (ii) above. That defines an equivalence relation on T, and we

[4] In (ii) and (iii), $s_\dagger$ is treated as a vector in $\mathbb{N}^{[|s_\dagger|] \times P}$.

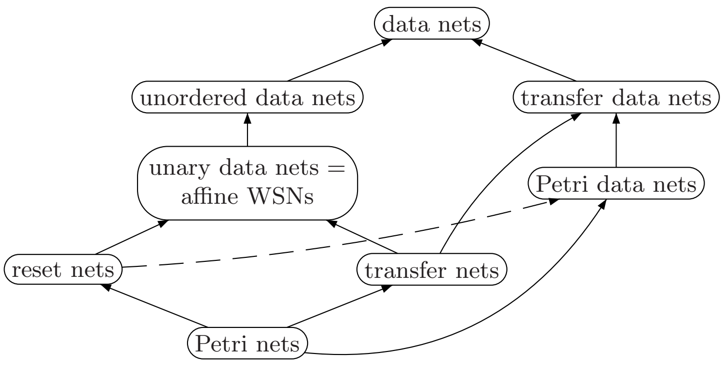


Fig. 1. Inclusions among classes of data nets

write $\overline{t}$ for the equivalence class of t. From the following proposition, the same-bag relation $\sim$ between markings is a bisimulation on the transition system of $\langle P, T, \alpha, F, G, H\rangle$.[5]

Proposition 4. *For any unordered data net, whenever $s_1 \xrightarrow{t} s_2$ and $s'_1 \sim s_1$, we have $s'_1 \xrightarrow{t'} s'_2$ for some $t' \sim t$ and $s'_2 \sim s_2$.*

Unary data nets. A data net $\langle P, T, \alpha, F, G, H\rangle$ is unary iff:

(i) for each $t \in T$, $\alpha_t = 1$;
(ii) for each $t \in T$, there exists $p \in P$ such that $F_t(1, p) > 0$;
(iii) for each $t \in T$, $R \in \mathit{Regs}(1)$ and $p, p' \in P$, we have $G_t(1, p, R, p') = 0$, $G_t(R, p, 1, p') = 0$, $G_t(R, p, R, p) = 1$, $G_t(R, p, R, p') = 0$ if $p \neq p'$, and $H_t(R, p) = 0$.

Proposition 5. *Any unary data net is an unordered data net.*

Given a unary data net $\mathcal{N} = \langle P, T, \alpha, F, G, H\rangle$, let $\widetilde{\mathcal{N}} = \langle P, T, \tilde{F}, \tilde{G}, \tilde{H}\rangle$ be the affine WSN such that $\tilde{F}$, $\tilde{G}$ and $\tilde{H}$ are obtained from F_t, G_t and H_t (respectively) by removing entries which involve indices from $\mathit{Regs}(1)$. Observe that, conversely, for each affine WSN $\mathcal{N}'$ in which no transition is firable from $\mathbf{0}$, there is a unique unary data net $\mathcal{N}$ such that $\widetilde{\mathcal{N}} = \mathcal{N}'$. Both $\mathcal{N} \mapsto \widetilde{\mathcal{N}}$ and its inverse are computable in logarithmic space.

[5] Conditions (i) and (ii) in the definition of unordered data nets suggest an alternative formalisation, where only one region is used for indexing F, G and H, and only one transition from each equivalence class is represented. Such a formalisation is more succinct (exponentially in transition arities), but that issue is not important in this paper. In addition, by Proposition 4, markings of unordered data nets can be regarded as bags.

Proposition 6. *(a) For any unary data net $\mathcal{N}$, we have that $s \xrightarrow{t} s'$ iff $|s'| = |s|$ and there exists $i \in [|s|]$ with $s(i) \xrightarrow{t} s'(i)$ in $\widetilde{\mathcal{N}}$ and $s'(j) = s(j)$ for all $j \neq i$.*
(b) Coverability of s' from s in a unary data net $\mathcal{N}$ is equivalent to existence of an increasing $\iota : [|s'|] \to [|s|]$ such that $s'(i)$ is coverable from $s(\iota(i))$ in $\widetilde{\mathcal{N}}$ for each $i \in [|s'|]$.
Termination (resp., boundedness) from s in a unary data net $\mathcal{N}$ is equivalent to $\widetilde{\mathcal{N}}$ being terminating (resp., bounded) from $s(i)$ for each $i \in [|s|]$.
(c) Coverability of m' from m, termination from m and boundedness from m in an affine well-structured net $\widetilde{\mathcal{N}}$ are equivalent to coverability of $\langle m' \rangle$ from $\langle m \rangle$, termination from $\langle m \rangle$ and boundedness from $\langle m \rangle$ (respectively) in $\mathcal{N}$.

Note that Proposition 6 (c) can be extended to affine WSN with transitions firable from $\mathbf{0}$ by adding an auxiliary place in which a single token is kept.

Transfer data nets. A data net $\langle P, T, \alpha, F, G, H \rangle$ is transfer iff:

(i) for each $t \in T$, $i \in [\alpha_t]$ and $p \in P$, we have $G_t(i, p, i', p') > 0$ for some $i' \in [\alpha_t]$ and $p' \in P$;
(ii) for each $t \in T$, $R \in \mathit{Regs}(\alpha_t)$ and $p \in P$, either we have $G_t(R, p, i', p') > 0$ for some $i' \in [\alpha_t]$ and $p' \in P$, or we have $G_t(R, p, R, p') > 0$ for some $p' \in P$.

Observe that (i) and (ii) are satisfied by the transition t in Example 1.

Proposition 7. *(a) Whenever $s_1 \xrightarrow{t} s_2$ in a data net and $s_1' \geq s_1$, there exists $s_2' \geq s_2$ such that $s_1' \xrightarrow{t} s_2'$.*
(b) Whenever $s_1 \xrightarrow{t} s_2$ in a transfer data net and $s_1' > s_1$, there exists $s_2' > s_2$ such that $s_1' \xrightarrow{t} s_2'$.

Petri data nets. In Petri data nets, whole-place operations are not allowed, and transitions can produce tokens carrying only data which were chosen during the firing. Formally, a data net $\langle P, T, \alpha, F, G, H \rangle$ is Petri iff:

– for each $t \in T$, $G_t = \mathit{Id}$;
– for each $t \in T$, $R \in \mathit{Regs}(\alpha_t)$ and $p \in P$, $H_t(R, p) = 0$.

Proposition 8. *Any Petri data net is a transfer data net.*

2.5 Example: A File System

As an illustration, we now show how a file system which permits unboundedly many users, user processes and files can be modelled as a data net. A variety of other examples of systems expressible using data nets can be found in [10,14,11,12,13,15,16], including a real-timed mutual exclusion protocol, a distributed authentication protocol, a communication protocol over unreliable channels, and a leader election algorithm.

We suppose there are two categories of users: administrators and staff members. Let `Administrator` be a finite set consisting of all possible states which an administrator process can be in, and let `Staff` be such a set for staff-member processes. (We assume that Administrator and Staff are disjoint.) We consider two file permissions, so let Permissions $= \{\texttt{private}, \texttt{public}\}$. We also suppose Contents is a finite set of all possible file contents. If file contents is unbounded, the Contents set may consist of finitary abstractions, which include information such as file names.

The set of places is

$$P = \texttt{Administrator} \cup \texttt{Staff} \cup (\texttt{Permissions} \times \texttt{Contents})$$

Tokens represent user processes and files, and data which they carry represents user identities. More specifically:

– a token at place $a \in \texttt{Administrator}$ carrying datum d represents a process of administrator d and which is in state a;
– a token at place $b \in \texttt{Staff}$ carrying datum d represents a process of staff member d and which is in state b;
– a token at place $\langle r, c\rangle \in \texttt{Permissions} \times \texttt{Contents}$ carrying datum d represents a file owned by user d, and with permission r and contents c.

To express a write by a staff-member process in state b to a file with contents c, which changes b to b' and c to c', we define a transition $\texttt{write}(b, b', c, c')$. It involves one user, so $\alpha_{\texttt{write}(b,b',c,c')} = 1$. Firstly, it takes one token from place b and one token from place c. They must carry the same datum, which ensures that the user owns the file.

$$F_{\texttt{write}(b,b',c,c')}(1, b) = 1 \qquad F_{\texttt{write}(b,b',c,c')}(1, c) = 1$$

The transition involves no whole-place operations, so $G_{\texttt{write}(b,b',c,c')} = Id$. Finally, it puts one token onto place b' and one token onto place c', which carry the same datum as the two tokens taken in the first stage.

$$H_{\texttt{write}(b,b',c,c')}(1, b') = 1 \qquad H_{\texttt{write}(b,b',c,c')}(1, c') = 1$$

The remaining entries of $F_{\texttt{write}(b,b',c,c')}$ and $H_{\texttt{write}(b,b',c,c')}$ are 0.

As a slightly more complex example, we can express a change of ownership of a file with permission r and contents c from an administrator to a staff member. It involves an administrator process which changes state from a to a', and a staff-member processes which changes state from b to b'. Since two users are involved, we have $\alpha_{\texttt{change}(r,c,a,a',b,b')} = 2$. As in the previous example, $G_{\texttt{change}(r,c,a,a',b,b')} = Id$ and we show only entries which are not 0:

$$\begin{array}{rr}
F_{\texttt{change}(r,c,a,a',b,b')}(1, \langle r, c\rangle) = 1 & H_{\texttt{change}(r,c,a,a',b,b')}(2, \langle r, c\rangle) = 1 \\
F_{\texttt{change}(r,c,a,a',b,b')}(1, a) = 1 & H_{\texttt{change}(r,c,a,a',b,b')}(1, a') = 1 \\
F_{\texttt{change}(r,c,a,a',b,b')}(2, b) = 1 & H_{\texttt{change}(r,c,a,a',b,b')}(2, b') = 1
\end{array}$$

In the $\texttt{change}(r, c, a, a', b, b')$ transition, it is assumed that the administrator identity is smaller than the staff-member identity. To cover the opposite case, and to have an unordered data net, we define a transition $\texttt{change}(r, c, b, b', a, a')$.

The definition is the same as that of $\texttt{change}(r, c, a, a', b, b')$, except that indices 1 and 2 are swapped when defining $F_{\texttt{change}(r,c,b,b',a,a')}$ and $H_{\texttt{change}(r,c,b,b',a,a')}$.

The data net having the three sets of transitions introduced so far is unordered and Petri. Implementing the following action makes it no longer Petri, in fact not even a transfer data net: all processes and files of a staff member who has a process which is in state b are removed from the system. We have $\alpha_{\texttt{crash}(b)} = 1$, $F_{\texttt{crash}(b)}(1, b) = 1$, the remaining entries of $F_{\texttt{crash}(b)}$ and all entries of $H_{\texttt{crash}(b)}$ are 0, and:

$$\begin{array}{rl}
G_{\texttt{crash}(s)}(1, p, 1, p') = 0 & \text{for } p, p' \in P \\
G_{\texttt{crash}(s)}(1, p, R, p') = 0 & \text{for } R \in \mathit{Regs}(1) \text{ and } p, p' \in P \\
G_{\texttt{crash}(s)}(R, p, 1, p') = 0 & \text{for } R \in \mathit{Regs}(1) \text{ and } p, p' \in P \\
G_{\texttt{crash}(s)}(R, p, R, p) = 1 & \text{for } R \in \mathit{Regs}(1) \text{ and } p \in P \\
G_{\texttt{crash}(s)}(R, p, R', p') = 0 & \text{otherwise}
\end{array}$$

Many interesting properties of the file system can be formalised as coverability, termination or boundedness properties. For example, that there is never a user who is both an administrator and a staff member amounts to none of the markings $s_{a,b}$ for $a \in \texttt{Administrator}$ and $b \in \texttt{Staff}$ being coverable, where $|s_{a,b}| = 1$, $s_{a,b}(1)(a) = s_{a,b}(1)(b) = 1$, and $s_{a,b}(1)(p) = 0$ for all $p \in P \setminus \{a, b\}$.

3 Reset Nets and Lossy Channel Systems

In this section, we first show how Petri data nets can express reset nets, which establishes the dashed inclusion in the diagram in Section 2.4. The translation preserves coverability and termination properties of reset nets.

Secondly, we show that Petri data nets can also express lossy channel systems [15]. The translation provides reductions of the location reachability and termination problems for lossy channel systems to the coverability, termination and boundedness problems for Petri data nets. Thus, the latter three problems will be shown non-primitive recursive: see Theorem 14.

Proposition 9. *(a) Coverability for reset nets is Turing reducible in polynomial space to coverability for Petri data nets.*
(b) Termination for reset nets is reducible in polynomial space to termination for Petri data nets, and to boundedness for Petri data nets.

Proof. We define a translation from reset nets $\mathcal{N} = \langle P, T, F, G, H\rangle$ to Petri data nets $\widehat{\mathcal{N}} = \langle \hat{P}, \hat{T}, \alpha, \hat{F}, \hat{G}, \hat{H}\rangle$. For each $t \in T$, let s_t^0 be a sequence consisting of all $p \in P$ which are reset by t, i.e., such that $G(p, p) = 0$ (each occurring once).

The set of places of $\widehat{\mathcal{N}}$ is formed by adding a place to P: $\hat{P} = P \uplus \{\hat{p}\}$. In $\widehat{\mathcal{N}}$, each place $p \in P$ will store a single token, carrying a datum which represents the place p of $\mathcal{N}$. The place $\hat{p}$ will store as many tokens carrying the datum which represents a place p as there are tokens at p in $\mathcal{N}$. More precisely, for markings m of $\mathcal{N}$ and s of $\widehat{\mathcal{N}}$, we write $m \approx s$ iff for each $p \in P$, there exists $j_p \in [|s|]$ such that: $s(j_p)(p) = 1$, $s(j')(p) = 0$ for all $j' \neq j_p$, and $s(j_p)(\hat{p}) = m(p)$. The relation $\approx$ will be a bisimulation between $\mathcal{N}$ and $\widehat{\mathcal{N}}$.

The transitions of $\widehat{\mathcal{N}}$ are pairs of transitions of $\mathcal{N}$ and enumerations of P: $\hat{T} = \{\hat{t}_\pi \,:\, t \in T \wedge [|P|] \overset{\pi}{\leftrightarrow} P\}$. Suppose $m \approx s$, and let π be the enumeration of P such that $\pi^{-1}(p) < \pi^{-1}(p')$ iff $j_p < j_{p'}$. We shall have that:

(i) only transitions of the form $\hat{t}_\pi$ are firable from s;

(ii) $m \xrightarrow{t} m'$ implies $s \xrightarrow{\hat{t}_\pi} s'$ for some $m' \approx s'$;

(iii) $s \xrightarrow{\hat{t}_\pi} s'$ implies $m \xrightarrow{t} m'$ for some $m' \approx s'$.

Consider any $\hat{t}_\pi \in \hat{T}$. We set $\alpha_{\hat{t}_\pi} = |P| + |s_t^0|$. Indices $i \in [|P|]$ will be used to pick data which represent the places of $\mathcal{N}$, and indices $|P| + i$ will be used to pick fresh data (which are greater than all existing data) to simulate the resets of t. Since $\hat{G}_{\hat{t}_\pi} = Id$ is required for $\widehat{\mathcal{N}}$ to be a Petri data net, it remains to define $\hat{F}_{\hat{t}_\pi}$ and $\hat{H}_{\hat{t}_\pi}$ so that (i)–(iii) above are satisfied. Each entry not listed below is set to 0:

$$\begin{array}{rrl}
\hat{F}_{\hat{t}_\pi}(i, \pi(i)) = 1 & \hat{F}_{\hat{t}_\pi}(i, \hat{p}) = F_t(\pi(i)) & (i \in [|P|]) \\
\hat{H}_{\hat{t}_\pi}(i, \pi(i)) = 1 & \hat{H}_{\hat{t}_\pi}(i, \hat{p}) = H_t(\pi(i)) & (\pi(i) \notin s_t^0) \\
\hat{H}_{\hat{t}_\pi}(|P| + i, s_t^0(i)) = 1 & \hat{H}_{\hat{t}_\pi}(|P| + i, \hat{p}) = H_t(s_t^0(i)) & (i \in [|s_t^0|])
\end{array}$$

Since any enumeration π of P is storable in polynomial space, we have that polynomial space suffices for the translation.

Given a marking m of $\mathcal{N}$, let s be a marking of $\widehat{\mathcal{N}}$ such that $m \approx s$. For (a), we have by (i)–(iii) above that a given marking m' is coverable from m in $\mathcal{N}$ iff some minimal s' such that $m' \approx s'$ is coverable from s in $\widehat{\mathcal{N}}$. For the first half of (b), we have by (i)–(iii) above that $\mathcal{N}$ terminates from m iff $\widehat{\mathcal{N}}$ terminates from s. For the second half, let $\widehat{\mathcal{N}}'$ be obtained from $\widehat{\mathcal{N}}$ (in logarithmic space) by adding a place $\hat{p}'$ and ensuring that each transition increases the number of tokens at $\hat{p}'$. Let s' be an arbitrary extension of s to place $\hat{p}'$. We have that $\mathcal{N}$ terminates from m iff $\widehat{\mathcal{N}}'$ is bounded from s'. $\square$

A *lossy channel system* is a tuple $\mathcal{S} = \langle Q, C, \Sigma, \Delta \rangle$, where Q is a finite set of locations, C is a finite set of channels, Σ is a finite alphabet, and $\Delta \subseteq Q \times C \times \{!, ?\} \times \Sigma \times Q$ is a set of transitions.

A state of $\mathcal{S}$ is a pair $\langle q, w \rangle$, where $q \in Q$ and $w : C \to \Sigma^*$. For each $c \in C$, the word $w(c)$ is the contents of channel c at state $\langle q, w \rangle$.

To define computation steps, we first define perfect computation steps, which either write a letter to the end of a channel, or read a letter from the beginning of a channel. For states $\langle q_1, w_1 \rangle$ and $\langle q_2, w_2 \rangle$, we write $\langle q_1, w_1 \rangle \rightarrow_{perf} \langle q_2, w_2 \rangle$ iff there exist $c \in C$ and $a \in \Sigma$ such that:

– either $\langle q_1, c, !, a, q_2 \rangle \in \Delta$ and $w_2 = w_1[c \mapsto (w_1(c))a]$,
– or $\langle q_1, c, ?, a, q_2 \rangle \in \Delta$ and $w_1 = w_2[c \mapsto a(w_2(c))]$.

Let $\sqsubseteq$ denote the "subword" well-quasi-ordering on Σ^*, obtained by lifting the equality relation on Σ (see Proposition 2). For example, we have $abba \sqsubseteq abracadabra$. For states $\langle q, w \rangle$ and $\langle q', w' \rangle$, we write $\langle q, w \rangle \sqsupseteq \langle q', w' \rangle$ iff $q = q'$ and $w(c) \sqsupseteq w'(c)$ for all $c \in C$, i.e., $\langle q', w' \rangle$ is obtained from $\langle q, w \rangle$ by losing zero or more letters.

A computation step $\langle q, w\rangle \to \langle q', w'\rangle$ of $\mathcal{S}$ consists of zero or more losses, followed by a perfect computation step, followed by zero or more losses. Thus, the $\to$ relation is defined by composing the $\to_{perf}$ and $\sqsupseteq$ relations: $\to\, = \,\sqsupseteq \to_{perf} \sqsupseteq$.

The following are two key decision problems for lossy channel systems:

Location reachability: Given a lossy channel system, a state $\langle q, w\rangle$ and a location q', to decide whether some state $\langle q', w'\rangle$ is reachable from $\langle q, w\rangle$.

Termination: Given a lossy channel system, and a state $\langle q, w\rangle$, to decide whether all computations from $\langle q, w\rangle$ are finite.

Proposition 10. *(a) Location reachability for lossy channel systems is reducible in logarithmic space to coverability for Petri data nets.*

(b) Termination for lossy channel systems is reducible in logarithmic space to termination for Petri data nets, and to boundedness for Petri data nets.

Proof. Given a lossy channel system $\mathcal{S} = \langle Q, C, \Sigma, \Delta\rangle$, we define a Petri data net $\mathcal{N}_\mathcal{S} = \langle P, T, \alpha, F, G, H\rangle$ as follows. We shall have that $\mathcal{N}_\mathcal{S}$ is computable in logarithmic space.

Let $P = Q \uplus C \uplus (C \times \Sigma)$. States $\langle q, w\rangle$ of $\mathcal{S}$ will be represented by markings $s \in Seq(\mathbb{N}^P \setminus \{\mathbf{0}\})$ as follows. At places in Q, there will be one token, which is at q, and which carries a datum d which is minimal in s. For each $c \in C$ with $w(c)$ empty, place c will contain one token which carries d. For each $c \in C$ with $w(c) = a_1 \cdots a_k$ and $k > 0$, there will be data $d \prec d_1^c \prec \cdots \prec d_k^c$ such that:

- place c contains one token which carries d_k^c;
- for each $a \in \Sigma$, place $\langle c, a\rangle$ contains one token carrying d_i^c for each $i \in [k]$ with $a_i = a$, and possibly some tokens carrying data greater than d_k^c.

Formally, we write $\langle q, w\rangle \approx s$ iff:

- $s(1)(q) = 1$, and $s(j)(q') = 0$ whenever either $j > 1$ or $q' \in Q \setminus \{q\}$;
- for each $c \in C$ with $w(c) = \varepsilon$, $s(1)(c) = 1$, and $s(j)(c) = 0$ for all $j > 1$;
- for each $c \in C$ with $w(c) = a_1 \cdots a_k$ and $k > 0$, there exist $1 < j_1^c < \cdots < j_k^c$ such that $s(j_k^c)(c) = 1$, $s(j')(c) = 0$ for all $j' \neq j_k^c$, and for each $1 \leq j' \leq j_k^c$ and $a' \in \Sigma$, we have

$$s(j')(c, a') = \begin{cases} 1, \text{if there exists } i \in [k] \text{ with } j' = j_i^c \text{ and } a' = a_i \\ 0, \text{otherwise} \end{cases}$$

For each read transition of $\mathcal{S}$, there will be $1+|\Sigma|$ transitions of $\mathcal{N}_\mathcal{S}$, depending on whether the channel will become empty after the read, or the last letter of the new channel contents will be a':

$$\begin{aligned} T = &\{\langle q_1, c, !, a, q_2\rangle \,:\, \langle q_1, c, !, a, q_2\rangle \in \Delta\} \cup \\ &\{\langle q_1, c, ?, a, q_2, \varepsilon\rangle, \langle q_1, c, ?, a, q_2, a'\rangle \,:\, \langle q_1, c, ?, a, q_2\rangle \in \Delta \wedge a' \in \Sigma\} \end{aligned}$$

When defining α_t, F_t and H_t for $t \in T$ below, we show only entries which are distinct from 0. Since $\mathcal{N}_\mathcal{S}$ is a Petri data net, we have $G_t = Id$ for each $t \in T$.

We shall have that, in computations of $\mathcal{N}_\mathcal{S}$, losses can happen only when reads are performed, but that will be sufficient for the result we are proving. Losses will occur when the datum which identifies the end of a channel and corresponds to the last letter is made smaller than the datum which corresponds to the second last letter. (Observe that, in data nets, we cannot specify that a transition be firable from a marking only if the latter contains no data which is between two particular data. If that were not so, perfect channel systems which are Turing-powerful would be expressible).

Writes are performed using the minimal datum, which is then decreased:

$$\begin{aligned} \alpha_{\langle q_1,c,!,a,q_2\rangle} &= 2 & H_{\langle q_1,c,!,a,q_2\rangle}(1,q_2) &= 1 \\ F_{\langle q_1,c,!,a,q_2\rangle}(2,q_1) &= 1 & H_{\langle q_1,c,!,a,q_2\rangle}(1,\langle c,a\rangle) &= 1 \end{aligned}$$

Reads which make a channel c empty alter the datum carried by the token at place c to be the minimal datum:

$$\begin{aligned} F_{\langle q_1,c,?,a,q_2,\varepsilon\rangle}(1,q_1) &= 1 & \alpha_{\langle q_1,c,?,a,q_2,\varepsilon\rangle} &= 2 \\ F_{\langle q_1,c,?,a,q_2,\varepsilon\rangle}(2,c) &= 1 & H_{\langle q_1,c,?,a,q_2,\varepsilon\rangle}(1,q_2) &= 1 \\ F_{\langle q_1,c,?,a,q_2,\varepsilon\rangle}(2,\langle c,a\rangle) &= 1 & H_{\langle q_1,c,?,a,q_2,\varepsilon\rangle}(1,c) &= 1 \end{aligned}$$

The remaining reads from channel c decrease the datum carried by the token at place c to a value which identifies an occurrence of some a':

$$\begin{aligned} F_{\langle q_1,c,?,a,q_2,a'\rangle}(1,q_1) &= 1 & \alpha_{\langle q_1,c,?,a,q_2,a'\rangle} &= 3 \\ F_{\langle q_1,c,?,a,q_2,a'\rangle}(3,c) &= 1 & H_{\langle q_1,c,?,a,q_2,a'\rangle}(1,q_2) &= 1 \\ F_{\langle q_1,c,?,a,q_2,a'\rangle}(3,\langle c,a\rangle) &= 1 & H_{\langle q_1,c,?,a,q_2,a'\rangle}(2,c) &= 1 \\ F_{\langle q_1,c,?,a,q_2,a'\rangle}(2,\langle c,a'\rangle) &= 1 & H_{\langle q_1,c,?,a,q_2,a'\rangle}(2,\langle c,a'\rangle) &= 1 \end{aligned}$$

Now, the definition of $\mathcal{N}_\mathcal{S}$ ensures that the $\approx$ relation is an inverse simulation: whenever $\langle q,w\rangle \approx s$ and $s \to s'$, there exists $\langle q',w'\rangle$ such that $\langle q',w'\rangle \approx s'$ and $\langle q,w\rangle \to \langle q',w'\rangle$.

We write $\langle q,w\rangle \sqsubseteq\approx s$ iff there exists $\langle q^\dagger,w^\dagger\rangle$ such that $\langle q,w\rangle \sqsubseteq \langle q^\dagger,w^\dagger\rangle$ and $\langle q^\dagger,w^\dagger\rangle \approx s$. It is straightforward to check that the $\sqsubseteq\approx$ relation is a simulation: whenever $\langle q,w\rangle \sqsubseteq\approx s$ and $\langle q,w\rangle \to \langle q',w'\rangle$, there exists s' such that $\langle q',w'\rangle \sqsubseteq\approx s'$ and $s \to s'$.

To establish (a), given a state $\langle q,w\rangle$ and a location q' of $\mathcal{S}$, let s be such that $\langle q,w\rangle \approx s$, and let s' be such that $|s'| = 1$, $s'(1)(q') = 1$, and $s'(1)(p) = 0$ for all $p \in P \setminus \{q'\}$. By the properties above, we have that some state $\langle q',w'\rangle$ is reachable from $\langle q,w\rangle$ iff some marking $s'' \geq s'$ is reachable from s.

For the termination part of (b), if s is such that $\langle q,w\rangle \approx s$, then $\mathcal{S}$ has an infinite computation from $\langle q,w\rangle$ iff $\mathcal{N}_\mathcal{S}$ has an infinite computation from s. For the boundedness part, we modify $\mathcal{N}_\mathcal{S}$ by adding an auxiliary place and ensuring that each transition increases the number of tokens at that place. $\square$

4 Decidability

The following two lemmas will be used in the proof of Theorem 13 below. The first one, due to Valk and Jantzen, provides a sufficient condition for computability of finite bases of upwards-closed sets of fixed-length tuples of natural numbers. The second lemma shows that, for computing a pred-basis of the upward closure of a marking of a data net, it suffices to consider markings up to a certain computable length.

Lemma 11 ([21]). *Suppose B is a finite set. A finite basis of an upwards-closed set $V \subseteq \mathbb{N}^B$ is computable iff it is decidable, given any $v \in \mathbb{N}^B_\omega$, whether $V \cap \downarrow\{v\} \neq \emptyset$.*

For a transition system $\langle S, \rightarrow\rangle$ and $S' \subseteq S$, we write $\mathit{Pred}(S')$ for $\{s \in S : \exists s' \in S' \cdot s \rightarrow s'\}$. If transitions are labelled by $t \in T$, we write $\mathit{Pred}_t(S')$ for $\{s \in S : \exists s' \in S' \cdot s \xrightarrow{t} s'\}$.

Lemma 12. *Given a data net $\mathcal{N}$, a transition t of $\mathcal{N}$, and a marking s' of $\mathcal{N}$, a natural number L is computable, such that whenever $s \in \mathit{Pred}_t(\uparrow\{s'\})$ and $|s| > L$, there exists $\bar{s} \leq s$ with $\bar{s} \in \mathit{Pred}_t(\uparrow\{s'\})$ and $|\bar{s}| \leq L$.*

Proof. Suppose $\mathcal{N} = \langle P, T, \alpha, F, G, H\rangle$, and let

$$L = \alpha_t + |s'| + (\alpha_t + 1) \times (2^{|P|} - 1) \times M$$

where $M = \max\{s'(i)(p) : i \in [|s'|] \wedge p \in P\}$.

Consider $s \in \mathit{Pred}_t(\uparrow\{s'\})$ with $|s| > L$. For some $s_\dagger$, ι and $s'' \geq s'$, we have $s \xrightarrow{t, s_\dagger, \iota} s''$. Let $s''_\dagger = (s_\dagger - [\![F_t]\!]_\iota^{|s_\dagger|})[\![G_t]\!]_\iota^{|s_\dagger|} + [\![H_t]\!]_\iota^{|s_\dagger|}$. Since s'' is the $\mathbf{0}$-contraction of $s''_\dagger$, there exists an increasing $\iota' : [|s'|] \rightarrow [|s_\dagger|]$ such that $s'(i) \leq s''_\dagger(\iota'(i))$ for all $i \in [|s'|]$.

For each nonempty $P_+ \subseteq P$, let

$$s_\dagger^{P_+} = \{i \in [|s_\dagger|] : \forall p \in P \cdot s_\dagger(i)(p) > 0 \Leftrightarrow p \in P_+\}$$

Since $|s_\dagger| \geq |s|$, there exist $0 \leq j \leq \alpha_t$ and nonempty $P_+ \subseteq P$ such that $|I_j^{P_+}| > M$, where $I_j^{P_+} = ([\![\mathit{Reg}_{(j,j+1)}]\!]_\iota^{|s_\dagger|} \setminus \mathit{Range}(\iota')) \cap s_\dagger^{P_+}$.

Pick an index $i^1_\dagger \in I_j^{P_+}$ of $s_\dagger$, and let $i^1 \in [|s|]$ be the corresponding index of s. Let $\tau_\dagger$ be the increasing mapping $[|s_\dagger| - 1] \rightarrow [|s_\dagger|]$ with $i^1_\dagger \notin \mathit{Range}(\tau_\dagger)$, and τ be the increasing mapping $[|s| - 1] \rightarrow [|s|]$ with $i^1 \notin \mathit{Range}(\tau)$. Then let $s^1_\dagger$ (resp., s^1) be obtained from $s_\dagger$ (resp., s) by removing the entry $i^1_\dagger$ (resp., i^1), $\iota_1 = \tau_\dagger^{-1} \circ \iota$, and $s''^1_\dagger = (s^1_\dagger - [\![F_t]\!]_{\iota_1}^{|s^1_\dagger|})[\![G_t]\!]_{\iota_1}^{|s^1_\dagger|} + [\![H_t]\!]_{\iota_1}^{|s^1_\dagger|}$. By the definition of $I_j^{P_+}$ and $|I_j^{P_+}| > M$, we have that $s''^1_\dagger(i)(p) \geq M$ whenever $s''^1_\dagger(i)(p) \neq s''_\dagger(\tau_\dagger(i))(p)$. Hence, $s''^1_\dagger \geq s'$, so $s^1 \in \mathit{Pred}_t(\uparrow\{s'\})$.

By repeating the above, we obtain $s \geq s^1 \geq s^2 \geq \cdots s^{|s|-L} \in \mathit{Pred}_t(\uparrow\{s'\})$ such that $|s^k| = |s| - k$ for all k. Setting $\bar{s} = s^{|s|-L}$ completes the proof. □

Theorem 13. *(a) Coverability and termination for data nets are decidable.*
(b) Boundedness for transfer data nets is decidable.

Proof. Suppose $\mathcal{N} = \langle P, T, \alpha, F, G, H\rangle$ is a data net. By Propositions 2, 3 and 7, we have that the transition system of $\mathcal{N}$ is finitely-branching and well-structured with strong compatibility, and also with strict compatibility if $\mathcal{N}$ is transfer (using the terminology of [18]). Moreover, $\leq$ between markings of $\mathcal{N}$ is a decidable partial ordering, and $\mathit{Succ}(s) = \{s' : s \rightarrow s'\}$ is computable for markings

s. Hence, termination for data nets and boundedness for transfer data nets are decidable by [18, Theorems 4.6 and 4.11].

To establish decidability of coverability by [18, Theorem 3.6], it suffices to show that, given any $t \in T$ and a marking s', a finite basis of $Pred_t(\uparrow\{s'\})$ is computable. (By Proposition 7 (a), $Pred_t(\uparrow\{s'\})$ is upwards-closed).

First, we compute L as in Lemma 12. For any $0 \leq l \leq L$, increasing $\eta : [l] \to [l_\dagger]$ and increasing $\iota : [\alpha_t] \to [l_\dagger]$ such that $[l_\dagger] = Range(\eta) \cup Range(\iota)$, let

$$Pred^l_{t,\eta,\iota}(\uparrow\{s'\}) = \{s \,:\, l = |s| \wedge \exists s'' \geq s' \cdot s \xrightarrow{t,\eta,\iota} s''\}$$

where $s \xrightarrow{t,\eta,\iota} s''$ means that $s \xrightarrow{t,s_\dagger,\iota} s''$ for some $s_\dagger$ such that $Range(\eta) = \{j : s_\dagger(j) \neq \mathbf{0}\}$ (necessarily, $l_\dagger = |s_\dagger|$). From the definition of transition firing, we have that $s \xrightarrow{t,s_\dagger,\iota} s''$ iff $s_\dagger \geq [\![F_t]\!]^{l_\dagger}_\iota$ and s'' is the $\mathbf{0}$-contraction of $(s_\dagger - [\![F_t]\!]^{l_\dagger}_\iota)[\![G_t]\!]^{l_\dagger}_\iota + [\![H_t]\!]^{l_\dagger}_\iota$. Hence, each $Pred^l_{t,\eta,\iota}(\uparrow\{s'\})$ is an upwards-closed subset of $\mathbb{N}^{P\times[l]}$. By Lemma 12, it remains to compute a finite basis of each $Pred^l_{t,\eta,\iota}(\uparrow\{s'\})$.

Suppose that l, η and ι are as above. Given any $s \in \mathbb{N}^{P\times[l]}_\omega$, we have as in [6] that $Pred^l_{t,\eta,\iota}(\uparrow\{s'\}) \cap \downarrow\{s\} \neq \emptyset$ iff $s_\dagger \geq [\![F_t]\!]^{l_\dagger}_\iota$ and $s'' \geq s'$, where $s_\dagger$ is the $\mathbf{0}$-expansion of s such that $l_\dagger = |s_\dagger|$ and $Range(\eta) = \{j : s_\dagger(j) \neq \mathbf{0}\}$, s'' is the $\mathbf{0}$-contraction of $(s_\dagger - [\![F_t]\!]^{l_\dagger}_\iota)[\![G_t]\!]^{l_\dagger}_\iota + [\![H_t]\!]^{l_\dagger}_\iota$, and the required operations are extended to ω by taking limits: $\omega \geq n$, $\omega + n = n + \omega = \omega + \omega = \omega$, $\omega - n = \omega$, $0 \times \omega = 0$, and $n \times \omega = \omega$ for $n > 0$. Therefore, by Lemma 11, a finite basis of $Pred^l_{t,\eta,\iota}(\uparrow\{s'\})$ is computable. □

5 Hardness

Theorem 14. *Coverability, termination and boundedness for Petri data nets are not primitive recursive.*

Proof. As shown in [7], location reachability and termination for lossy channel systems are not primitive recursive. It remains to apply Proposition 10. □

Theorem 15. *Coverability, termination and boundedness for unordered Petri data nets are not elementary.*

Proof. For $n \in \mathbb{N}$, the *tetration* operation $a \Uparrow n$ is defined by $a \Uparrow 0 = 1$ and $a \Uparrow (n+1) = a^{a \Uparrow n}$.
The non-elementariness of the three verification problems follows from showing that, given a deterministic machine $\mathcal{M}$ of size n with finite control and two $2 \Uparrow n$-bounded counters, an unordered Petri data net $\mathcal{N}_\mathcal{M}$ which simulates $\mathcal{M}$ is constructible in logarithmic space. A counter is m-bounded iff it can have values in $\{0, \ldots, m-1\}$, i.e., it cannot be incremented beyond the maximum value $m-1$. The following counter operations may be used in $\mathcal{M}$: increment, decrement, reset, iszero and ismax.

It will be defined below when a marking of $\mathcal{N}_\mathcal{M}$ represents a configuration (i.e., state) of $\mathcal{M}$. Let us call such markings "clean". We write $s \to_\surd s'$ (resp.,

$s \rightarrow_{\times} s'$) iff $s \rightarrow s'$ and s' is clean (resp., not clean). Hence, $s \rightarrow_{\times}^{*}\rightarrow_{\surd} s'$ means that s' is clean and reachable from s by a nonempty sequence of transitions in which every intermediate marking is not clean, and $s \not\rightarrow_{\times}^{\omega}$ means that there does not exist an infinite sequence of transitions from s in which no intermediate marking is clean. $\mathcal{M}$ will be simulated in the following sense by $\mathcal{N}_{\mathcal{M}}$ from a certain initial marking s_I, where c_I is the initial configuration of $\mathcal{M}$:

– we have $s_I \not\rightarrow_{\times}^{\omega}$ and:
  - there exists $s_I \rightarrow_{\times}^{*}\rightarrow_{\surd} s'$ such that c_I is represented by s';
  - for all $s_I \rightarrow_{\times}^{*}\rightarrow_{\surd} s'$, c_I is represented by s';

– whenever c is represented by s, we have $s \not\rightarrow_{\times}^{\omega}$ and:
  - if c has a successor c', there exists $s \rightarrow_{\times}^{*}\rightarrow_{\surd} s'$ with c' represented by s';
  - for all $s \rightarrow_{\times}^{*}\rightarrow_{\surd} s'$, c has a successor c' which is represented by s'.

That $\mathcal{M}$ halts (i.e. reaches a halting control state from c_I) will therefore be equivalent to a simple coverability question from s_I, and to termination from s_I. After extending $\mathcal{N}_{\mathcal{M}}$ by a place whose number of tokens increases with each transition, that $\mathcal{M}$ halts becomes equivalent to boundedness from s_I.

Each clean marking s of $\mathcal{N}_{\mathcal{M}}$ will represent a valuation v of $3n$ counters C_k, C'_k and C''_k for $k \in [n]$. C_n and C'_n are the two counters of $\mathcal{M}$, and for each $k \in [n]$, C_k, C'_k and C''_k are $2 \Uparrow k$-bounded. (Counter C''_n will not be used, so it can be omitted.) $\mathcal{N}_{\mathcal{M}}$ will have places 0_D, 1_D, $\mathit{scratch}_D$, lock_D, $\mathit{checked}_D$ and $\mathit{unchecked}_D$ for each $D \in \{C_k, C'_k, C''_k : k \in [n]\}$, as well as a number (polynomial in n) of places for encoding the control of $\mathcal{M}$ and for control of $\mathcal{N}_{\mathcal{M}}$. A valuation v is represented by s as follows:

– for each $k \in [n]$ and $D \in \{C_k, C'_k, C''_k\}$, places $\mathit{scratch}_D$, lock_D and $\mathit{checked}_D$ are empty, and $\mathit{unchecked}_D$ contains exactly $2 \Uparrow (k-1)$ tokens and they carry mutually distinct data;

– for each $k \in [n]$, $D \in \{C_k, C'_k, C''_k\}$ and $i \in [2 \Uparrow (k-1)]$, if the i-th bit of $v(D)$ is $b \in \{0,1\}$, then for some datum d carried by a token at place $\mathit{unchecked}_D$, the number of tokens at b_D which carry d is i, and the number of tokens at $(1-b)_D$ which carry d is 0;

– for each $k \in [n]$ and $D \in \{C_k, C'_k, C''_k\}$, each datum carried by a token at 0_D or 1_D is carried by some token at $\mathit{unchecked}_D$.

Counters C_1, C'_1 and C''_1 are 2-bounded, so operations on them are trivial to simulate. For each $k < n$, counter operations on C_{k+1}, C'_{k+1} and C''_{k+1} are simulated using operations on C_k, C'_k and C''_k. The following shows how to implement $\mathsf{iszero}(D)$, where $D \in \{C_{k+1}, C'_{k+1}, C''_{k+1}\}$. The other four counter operations are implemented similarly.

```
for $C_k := 0$ to $(2 \Uparrow k) - 1$ do
{ guess a datum $d$ and move a token carrying $d$ from $\mathit{unchecked}_D$ to $\mathit{lock}_D$;
  for $C'_k := 0$ to $C_k$ do { move a token carrying $d$ from $0_D$ to $\mathit{scratch}_D$ };
  for $C'_k := 0$ to $C_k$ do { move a token carrying $d$ from $\mathit{scratch}_D$ to $0_D$ };
  move the token from $\mathit{lock}_D$ to $\mathit{checked}_D$ };
for $C_k := 0$ to $(2 \Uparrow k) - 1$ do
{ move a token from $\mathit{checked}_D$ to $\mathit{unchecked}_D$ }
```

Observe that $\mathsf{iszero}(D)$ can execute completely iff, for each $i \in [2 \Uparrow k]$, the datum d guessed in the i-th iteration of the outer loop represents the i-th bit of $v(D)$ and that bit is 0. Place $lock_D$ is used for keeping the datum d during each such iteration, and it is implicitly employed within the two inner loops.

It remains to implement routines $\mathsf{setup}(D)$ for $k \in [n]$ and $D \in \{C_k, C'_k, C''_k\}$, which start from empty 0_D, 1_D, $scratch_D$, $lock_D$, $checked_D$ and $unchecked_D$, and set up 0_D and $unchecked_D$ to represent D having value 0. Setting up C_1, C'_1 and C''_1 is trivial. To implement $\mathsf{setup}(D)$ for $k < n$ and $D \in \{C_{k+1}, C'_{k+1}, C''_{k+1}\}$, we use C_k, C'_k and C''_k which were set up previously. The implementation is similar to that of $\mathsf{iszero}(D)$, except that all three of C_k, C'_k and C''_k are used, since whenever a datum d is picked to be the i^{th} datum at $unchecked_D$ for some $i \in [2 \Uparrow k]$, two nested loops are employed to ensure that d is distinct from each of $i - 1$ data which are carried by tokens already at $unchecked_D$. □

6 Concluding Remarks

We have answered questions (1) and (2) posed in Section 1. As far as we are aware, Section 5 contains the first nontrivial lower bounds on complexity of decidable problems for extensions of Petri nets by infinite data domains.

The results obtained and their proofs show that data nets are a succinct unifying formalism which is close to the underlying semantic structures, and thus a useful platform for theoretical investigations.

The proof of Theorem 13 does not provide precise upper bounds on complexity. It should be investigated whether upper bounds which match the lower bounds in the proofs of Theorems 14 and 15 are obtainable. In particular, are coverability, termination and boundedness for unordered Petri data nets primitive recursive?

Let us say that a data net is l, m*-safe* iff each place other than some l places never contains more than m tokens. It is not difficult to tighten the proofs of Theorems 14 and 15 to obtain that coverability, termination and boundedness are not primitive recursive for $1, 1$-safe Petri data nets, and not elementary for $2, 1$-safe unordered Petri data nets. That leaves open whether we have non-elementarity for $1, 1$-safe unordered Petri data nets. That class suffices for expressing polymorphic systems with one array of type $\langle X, = \rangle \to \langle Y, = \rangle$ without whole-array operations [16,17].

We are grateful to Alain Finkel for a helpful discussion.

References

1. Reisig, W.: Petri Nets: An Introduction. Springer, Heidelberg (1985)
2. Girault, C., Valk, R. (eds.): Petri Nets for Systems Engineering. Springer, Heidelberg (2003)
3. Esparza, J., Nielsen, M.: Decidability issues for Petri nets – a survey. Bull. EATCS 52, 244–262 (1994)
4. Lipton, R.J.: The reachability problem requires exponential space. Technical Report 62, Yale University (1976)
5. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. 6, 223–231 (1978)

6. Finkel, A., McKenzie, P., Picaronny, C.: A well-structured framework for analysing Petri net extensions. Inf. Comput. 195(1–2), 1–29 (2004)
7. Schnoebelen, P.: Verifying lossy channel systems has nonprimitive recursive complexity. Inf. Proc. Lett. 83(5), 251–261 (2002)
8. Odifreddi, P.: Classical Recursion Theory II. Elsevier, Amsterdam (1999)
9. 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)
10. Abdulla, P.A., Nylén, A.: Timed Petri nets and BQOs. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 53–70. Springer, Heidelberg (2001)
11. Delzanno, G.: Constraint multiset rewriting. Technical Report DISI-TR-05-08, Università di Genova Extends [22–24] (2005)
12. Abdulla, P.A., Delzanno, G.: Constrained multiset rewriting. In: AVIS. ENTCS 2006 (to appear 2006)
13. Rosa Velardo, F., de Frutos Escrig, D., Marroquín Alonso, O.: On the expressiveness of mobile synchronizing Petri nets. In: SECCO. ENTCS 2005 (to appear 2005)
14. Abdulla, P.A., Jonsson, B.: Model checking of systems with many identical timed processes. Theor. Comput. Sci. 290(1), 241–264 (2003)
15. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. 127(2), 91–101 (1996)
16. Lazić, R., Newcomb, T.C., Roscoe, A.W.: Polymorphic systems with arrays, 2-counter machines and multiset rewriting. In: Infinity '04, ENTCS, vol. 138, pp. 61–86 (2005)
17. Lazić, R.: Decidability of reachability for polymorphic systems with arrays: A complete classification. In: Infinity '04, ENTCS, vol. 138, pp. 3–19 (2005)
18. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere? Theor. Comput. Sci. 256(1–2), 63–92 (2001)
19. Meyer, A.R.: Weak monadic second-order theory of successor is not elementary-recursive. In: Logic colloquium '72–73. Lect. Not. Math, vol. 453, pp. 132–154. Springer, Heidelberg (1975)
20. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. (3) 2(7), 326–336 (1952)
21. Valk, R., Jantzen, M.: The residue of vector sets with applications to decidability problems in Petri nets. Acta Inf. 21, 643–674 (1985)
22. Delzanno, G.: An assertional language for systems parametric in several dimensions. In: VEPAS, ENTCS, vol. 50 (2001)
23. Bozzano, M., Delzanno, G.: Beyond parameterized verification. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 221–235. Springer, Heidelberg (2002)
24. Bozzano, M., Delzanno, G.: Automatic verification of invalidation-based protocols. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 295–308. Springer, Heidelberg (2002)

Operating Guidelines for Finite-State Services[*]

Niels Lohmann[1], Peter Massuthe[1], and Karsten Wolf[2]

[1] Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
{nlohmann,massuthe}@informatik.hu-berlin.de
[2] Universität Rostock, Institut für Informatik,
18051 Rostock, Germany
karsten.wolf@informatik.uni-rostock.de

Abstract. We study services modeled as *open workflow nets* (oWFN) and describe their behavior as *service automata*. Based on arbitrary finite-state service automata, we introduce the concept of an *operating guideline*, generalizing the work of [1,2] which was restricted to acyclic services.

An operating guideline gives complete information about how to properly interact (in this paper: deadlock-freely and with limited communication) with an oWFN N. It can be executed, thus forming a properly interacting partner of N, or it can be used to support service discovery.

An operating guideline for N is a particular service automaton S that is enriched with Boolean annotations. S interacts properly with the service automaton *Prov*, representing the behavior of N, and is able to simulate every other service that interacts properly with *Prov*. The attached annotations give complete information about whether or not a simulated service interacts properly with *Prov*, too.

1 Introduction

In real life, we routinely use complicated electronic devices such as digital cameras, alarm clocks, mobile phones, CD players, vending machines, etc. Using such a device involves complex interaction, where information from the user to the device flows via pushing buttons or spinning wheels while information is passed from the device to the user via displays or blinking LEDs.

In some cases, we do not even abstractly know what is going on inside the device. Nevertheless, we are typically able to participate in the interaction. Besides ergonomic design, help from experienced friends, or trial-and-error exploration, it is often the user instructions which help us to figure out what to do at which stage. The typical features of user instructions (at least good ones) are:

- they are shipped with, or pinned to, the device,
- they are operational, that is, a user can execute them step by step,
- they are complete, that is, they cover the full intended functionality of the device,

[*] Partially funded by the BMBF project "Tools4BPEL".

– they use only terms related to the interface (buttons, displays, etc.) without trying to explain the internal processes.

In the virtual world, services [3] replace the devices of the real world. Still, using a service may require involved interaction with the user (which can be another service, like in service-oriented computing [4]). With the concept of an *operating guideline*, we are going to propose an artifact that, in the virtual world, plays the role of user instructions in the real world. In particular, we will show that it exhibits the characteristics listed above. Moreover, we show that the operating guideline for a service can be automatically computed and be used for automatically checking proper interaction between services.

In contrast, a *public view* of a service (a condensed version of the service itself) has been proposed as another artifact for explaining the interaction with the service [5,6]. Public views, however, do neither match the second nor the fourth item of the list above.

Our approach is based on the behavior of *open workflow nets* (oWFN) [2]. oWFN are a class of Petri nets which has been proposed for modeling services. oWFN generalize and extend the classical *workflow nets* [7]. The most important extension is an interface for asynchronous message passing. This interface allows us to compose services to larger units. Suitability of oWFN for modeling services has been proven through an implemented translation from the industrial service description language WS-BPEL [8] into oWFN [9,10]. While there are many partial formalizations for WS-BPEL, the translation to oWFN is feature complete. Other feature complete formalizations are based on *abstract state machines* [11,12].

We describe the *behavior* of an oWFN with the help of a *service automaton*. A service automaton basically records the internal states of an oWFN. The transitions of the automaton are labeled with information about message passing through the mentioned interface. Service automata form the basis of the proposed *operating guidelines*.

Operating guidelines have so far been introduced for *acyclic services* [1,2]. In this paper, we extend our previous results and introduce the concept of an operating guideline for an *arbitrary finite-state* service N. The operating guideline of N is a distinguished service automaton S that properly interacts with N, together with *Boolean annotations* at each state of S. The annotations serve as a characterization of *all* services that properly interact with N.

For this paper, we assume that "proper interaction" between services N and N' means deadlock freedom of the system composed of N and N' and limited communication, that is, k-boundedness of all message buffers, for some given k. We are well aware that there are other possibilities for defining "proper interaction". Nevertheless, deadlock freedom and limited communication will certainly be part of any such definition, so this paper can be seen as a step towards a more sophisticated setting.

The rest of the paper is organized as follows. In Sect. 2, we introduce open workflow nets, service automata, and their relation. Sections 3 to 5 are devoted to the construction of an operating guideline and its use. These sections build entirely upon the concept of service automata. We start by defining, in Sect. 3, a concept that we call *situations*. This concept is fundamental to our approach. It describes the interaction of a given service automaton *Prov* with a partner *Req* from the point of view of *Req* only. With the help of situations, we are able to characterize deadlock freedom of the interaction in a way that is suitable for subsequent considerations. The characterization is translated into Boolean formulas which are used as annotations in the operating guideline later on. The calculation and justification of the canonical partner S mentioned above is subject of Sect. 4. In Sect. 5, finally, we formalize the concept of an operating guideline and show how it can be used for identifying other partners that communicate deadlock-freely with *Prov*. Section 6 discusses issues of an implementation and presents experimental results. Finally, we summarize the results of the paper and sketch our plans for further work.

2 Models for Services

2.1 Open Workflow Nets (oWFN)

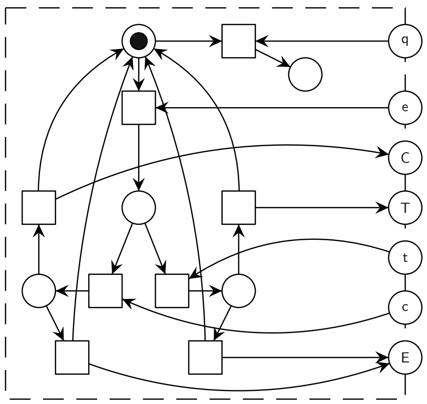
The introduction of open workflow nets [2] was inspired by the view of a service as a workflow plus an interface. Consequently, oWFN extend workflow nets [7] with an interface for asynchronous message passing. oWFN further drop some syntactic restrictions present in workflow nets, for instance the unique start and end places. These restrictions would complicate service composition without sufficient justification by improved analysis possibilities.

Definition 1 (Open workflow net). *An* open workflow net *consists of:*

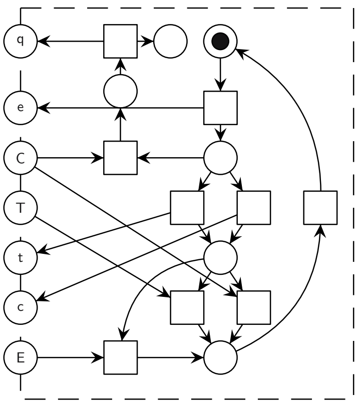
- *an ordinary place/transition net* $[P, T, F, m_0]$*; together with*
- *two disjoint sets* $P_i, P_o \subseteq P$*, called* input *and* output *places, such that* $F \cap (P_o \times T) = \emptyset$ *and* $F \cap (T \times P_i) = \emptyset$*. We assume* $m_0(p) = 0$ *for* $p \in P_i \cup P_o$*;*
- *a set* Ω *of markings, called* final markings*. For* $m_f \in \Omega$ *and* $p \in P_i \cup P_o$*, we assume* $m_f(p) = 0$*. We further require that a marking in* m_f *does not enable a transition.*

P_i represents channels for incoming messages, P_o channels for outgoing messages. The required restrictions for arcs guarantee that sent messages cannot be "unsent" and received messages cannot be "unreceived". Our set of final markings replaces the single final place of workflow nets. Any marking may be final as long as it does not enable a transition. That is, a service does not perform actions in a final marking. A service may, however, be designed such that it resumes work when it receives a message while residing in a final marking.

A major intention behind services is their composition to larger units. Correspondingly, there is a concept of composition for oWFN. oWFN are composed by merging interface places. This can, in general, be done for arbitrarily many oWFN. For the purpose of this paper, it is sufficient to understand the composition of just two oWFN.



(a) A vending machine oWFN N_V.



(b) A partner oWFN of N_V.

Fig. 1. The oWFN N_V (a) models a vending machine. In every iteration, it first expects a coin to be inserted (message e), and a choice for a coffee or a tea to be made (c or t). It returns either the coin (E), modeling a failed check for validity, or the corresponding beverage (C or T). The machine can be shut down by sending q. The oWFN (b) models a potential partner (user) of the vending machine.

Definition 2 (Partner oWFN). *Two oWFN N and N' are partners if $P_i = P'_o$ and $P_o = P'_i$. All other ingredients of N and N' are assumed to be disjoint.*

If two oWFN are partners, they can be composed. Composition consists mainly of merging the interface places.

Definition 3 (Composition of oWFN). *Let N and N' be partner oWFN. The composition $m \oplus m' : P \cup P' \to \mathbb{N}$ of two markings m of N and m' of N' is defined by $(m \oplus m')(p) = m(p)$, if $p \in P$ and $(m \oplus m')(p) = m'(p)$, if $p \in P'$. The* composition of N and N' *is the oWFN $N \oplus N'$ defined as follows:*

- $P_{N\oplus N'} = P \cup P'$, $T_{N\oplus N'} = T \cup T'$, $F_{N\oplus N'} = F \cup F'$, $m_{0_{N\oplus N'}} = m_0 \oplus m'_0$;
- $P_{i_{N\oplus N'}} = P_{o_{N\oplus N'}} = \emptyset$;
- $\Omega_{N\oplus N'} = \{m \oplus m' \mid m \in \Omega, m' \in \Omega'\}$.

The composition of markings is well-defined for partners, as the common places of N and N' do not carry tokens. The composition of partners leads to an oWFN with an empty interface. As an example, Fig. 1 shows two partner oWFN.

Only for oWFN with empty interface it is reasonable to consider their reachability graph (occurrence graph), as an interface suggests some interaction with a (possibly unknown) environment. We rely on the usual concept of reachability graph. For studying a service in isolation, we consider the *inner* of an (uncomposed) oWFN. The inner of N is an oWFN with empty interface, so its reachability graph may be considered.

Definition 4 (Inner). *Let N be an oWFN. The* inner of N*, denoted* $inner(N)$*, is obtained from N by removing all places in P_i and P_o and their adjacent arcs. Initial and final markings are adjusted accordingly.*

This leads to the following definition of boundedness of arbitrary oWFN.

Definition 5 (Boundedness of oWFN). *An oWFN N is* bounded *if the reachability graph of $inner(N)$ is finite.*

Boundedness, as defined above, concerns the inner of an oWFN. The composition of two bounded oWFN, however, can still be unbounded since tokens may be accumulated in the merged interface places. Thus, we have to introduce an additional concept of boundedness of the interface places.

Definition 6 (Limited communication of oWFN). *Two partner oWFN N and N' have k-*limited communication *(for some $k \in \mathbb{N}$) if $m(p) \leq k$ for all markings m reachable in $N \oplus N'$ and all places $p \in P_i \cup P'_i$.*

If two bounded oWFN N and N' are k-limited partners, then $N \oplus N'$ is bounded, too.

2.2 Service Automata

Open workflow net models as introduced so far can be obtained from practical specifications of services. There is, for instance, a feature complete translation from WS-BPEL to oWFN [9,10].

In this section we introduce *service automata* [1], which serve as the basis of the calculation of operating guidelines. A state of a service automaton is comparable to a marking of the inner of an oWFN. Communication activities are modeled as annotations to the transitions of a service automaton.

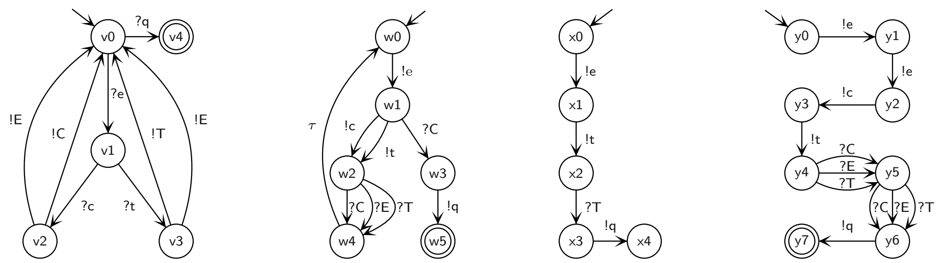
Service automata differ from standard I/O-automata [13]. They communicate asynchronously rather than synchronously, and they do not require explicit modeling of the state of the message channels. This approach leads to smaller and thus more readable automata. Other versions of automata models for services were proposed by [14] and [3], for instance. [14] model communication as occurrences of labels with no explicit representation of pending messages, whereas [3] use bounded and unbounded queues to store such messages.

Unlike an oWFN, a single service automaton has no explicit concept of message channels. The channels are taken care of in the definition of composition: a state of a composed service automaton consists of a state of each participating service automaton and a state of the *message bag* of currently pending messages.

We fix a finite set C, the elements of which we call *channels*. They take the role of the interface places in oWFN. We assume $\tau \notin C$ (the symbol τ is reserved for an internal move). With $bags(C)$, we denote the set of all multisets over C, that is, all mappings $m : C \to \mathbb{N}$. A multiset over C models a state of the message bag, that is, it represents, for each channel, the number of pending messages. $[\,]$ denotes the empty multiset ($[\,](x) = 0$ for all x), $[x]$ a singleton multiset ($[x](x) = 1$, $[x](y) = 0$ for $y \neq x$), $m_1 + m_2$ denotes the sum of two multisets ($(m_1 + m_2)(x) = m_1(x) + m_2(x)$ for all x), and $m_1 - m_2$ the difference ($(m_1 - m_2)(x) = max(m_1(x) - m_2(x), 0)$ for all x). $bags_k(C)$ denotes the set of all those multisets m over C where $m(x) \leq k$ for all x. $bags_k(C)$ is used for modeling the concept of limited communication.

Definition 7 (Service automata). *A* service automaton $A = [Q, I, O, \delta, q_0, F]$ *consists of a set Q of* states, *a set $I \subseteq C$ of* input channels, *a set $O \subseteq C$, $I \cap O = \emptyset$ of* output channels, *a nondeterministic* transition relation $\delta \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$, *an* initial state $q_0 \in Q$, *and a set of* final states $F \subseteq Q$ *such that $q \in F$ and $[q, x, q'] \in \delta$ implies $x \in I$. A is* finite *if its set of states is finite.*

Throughout this paper, we use the following notations for service automata. With *Prov* (from service *prov*ider), we denote an arbitrary service automaton for which we are going to calculate its operating guideline. With *Req* (from service *req*uester), we denote an arbitrary service automaton in its role as a communication partner of *Prov*. S is used for the particular partner of *Prov* that forms the core of the operating guideline for *Prov*. Service automata without an assigned role are denoted A. We use indices to distinguish the constituents of different service automata. In figures, we represent a channel $x \in I$ with $?x$ and a channel $y \in O$ with $!y$. Figure 2 shows four examples of service automata.



(a) V (b) W (c) X (d) Y

Fig. 2. Examples of service automata. The service automaton V models our vending machine (see Fig. 1(a)). The service automata W, X, and Y model partners of V. Final states are depicted by double circles.

Definition 8 (Partner automata). *Two service automata A and B are* partner *automata if $I_A = O_B$ and $I_B = O_A$.*

As in the case of oWFN, partner automata can be composed.

Definition 9 (Composition of service automata). *For partner automata A and B, their* composition *is defined as the service automaton $A \oplus B = [Q_{A \oplus B}, I_{A \oplus B}, O_{A \oplus B}, \delta_{A \oplus B}, q_{0_{A \oplus B}}, F_{A \oplus B}]$ defined as follows:*

$Q_{A \oplus B} = Q_A \times Q_B \times bags(C)$, $I_{A \oplus B} = O_{A \oplus B} = \emptyset$, $q_{0_{A \oplus B}} = [q_{0_A}, q_{0_B}, [\,]]$, $F_{A \oplus B} = F_A \times F_B \times \{[\,]\}$. The transition relation $\delta_{A \oplus B}$ contains the elements

- $[[q_A, q_B, m], \tau, [q'_A, q_B, m]]$ *iff* $[q_A, \tau, q'_A] \in \delta_A$ *(internal move in A),*
- $[[q_A, q_B, m], \tau, [q_A, q'_B, m]]$ *iff* $[q_B, \tau, q'_B] \in \delta_B$ *(internal move in B),*
- $[[q_A, q_B, m], \tau, [q'_A, q_B, m - [x]]]$ *iff* $[q_A, x, q'_A] \in \delta_A$*,* $x \in I_A$*, and* $m(x) > 0$ *(receive by A),*
- $[[q_A, q_B, m], \tau, [q_A, q'_B, m - [x]]]$ *iff* $[q_B, x, q'_B] \in \delta_B$*,* $x \in I_B$*, and* $m(x) > 0$ *(receive by B),*
- $[[q_A, q_B, m], \tau, [q'_A, q_B, m + [x]]]$ *iff* $[q_A, x, q'_A] \in \delta_A$ *and* $x \in O_A$ *(send by A),*
- $[[q_A, q_B, m], \tau, [q_A, q'_B, m + [x]]]$ *iff* $[q_B, x, q'_B] \in \delta_B$ *and* $x \in O_B$ *(send by B),*

and no other elements.

Figure 3 depicts the composition $V \oplus W$ of the services V and W of Fig. 2.

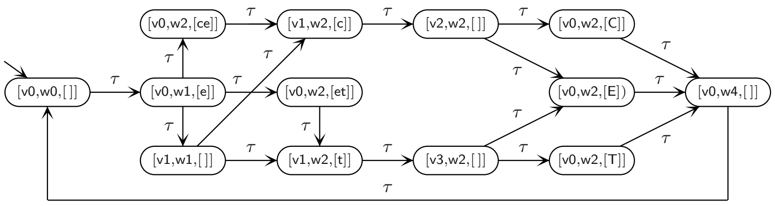


Fig. 3. The composed system $V \oplus W$ of the service automata V and W of Fig. 2. Only states reachable from the initial state are depicted. Note that $V \oplus W$ has no (reachable) final states. Nevertheless, $V \oplus W$ is deadlock-free, which is central in this paper.

Definition 10 (Wait state, deadlock). *For an automaton A, a state q is called a* wait state *iff* $[q, x, q'] \in \delta$ *implies* $x \in I$*, that is, q cannot be left without help from the environment. For a wait state q, let* $wait(q) = \{x \in I \mid \exists q' \in Q : [q, x, q'] \in \delta\}$*. A wait state q is called* deadlock *iff* $q \notin F$ *and* $wait(q) = \emptyset$*.*

A wait state cannot be left without an incoming message. $wait(q)$ is the set of all incoming messages that would help to leave q. A deadlock cannot be left, independently from incoming messages. The definition of service automata requires final states to be wait states which is reasonable.

Examples for wait states in Fig. 2 are v0 with $wait(\mathsf{v0}) = \{\mathsf{e}, \mathsf{q}\}$, w2 with $wait(\mathsf{w2}) = \{\mathsf{C}, \mathsf{E}, \mathsf{T}\}$, or x4 with $wait(\mathsf{x4}) = \emptyset$. An example for a deadlock is the state $[\mathsf{v0}, \mathsf{x2}, [\mathsf{E}]]$ of the (not depicted) composition of the services V and X of Fig. 2 that can be reached from the initial state $[\mathsf{v0}, \mathsf{x0}, [\,]]$ of $V \oplus X$ by executing first the transitions send e and send t of service X, followed by the transitions receive e, receive t, and send E of service V.

For service automata, limited communication can be formalized as follows.

Definition 11 (Limited communication of service automata). *Let A and B be two partner automata and* $A \oplus B$ *their composition. Then, A is called a* k-limited communication partner *of B iff* $Q_{A \oplus B} \subseteq Q_A \times Q_B \times bags_k(C)$*.*

Throughout the paper, we assume k to be given and fixed. The value of k may be chosen either by considerations on the physical message channels, by a static analysis that delivers a "sufficiently high" value, or just randomly. If two finite service automata *Prov* and *Req* are k-limited partners, then $Prov \oplus Req$ is finite as well. In Fig. 2, W and X are 1-limited partners of V. Y is no 1-limited partner since $V \oplus Y$ contains, for instance, the state $[\mathsf{v0}, \mathsf{y2}, [\mathsf{ee}]]$. Y is, however, a 2-limited partner of V.

Every k-limited communication partner is a $(k+1)$-limited communication partner as well. For every value of k, there are services which have a deadlock-free k-limited communication partner but no deadlock-free $(k-1)$-limited communication partner. There are even services which have deadlock-free communication partners but not a k-limited one for any k. As an example, consider the service in Fig. 5(a): A communication partner would have a single (initial and final) state s in which it receives a and loops back to s.

2.3 Translation from oWFN to Service Automata

While there exist direct translations from WS-BPEL to automata and closely related formalisms [15,16,17] we propose to generate service automata from oWFN. This way, we can directly inherit the already mentioned feature completeness of the Petri net translations of WS-BPEL [9,10].

Comparing the behavior of oWFN and service automata, the main difference is the capability of oWFN to send and receive several messages in a single transition occurrence. In order to keep matters simple, we give a direct translation from an oWFN N to a service automaton only for the case that every transition of N is connected to at most one place in $P_i \cup P_o$. In fact, this assumption holds for all oWFN stemming from WS-BPEL specifications as a BPEL activity accesses at most one message channel. On the other hand, an arbitrary oWFN can be transformed in various ways to match the requirement. We sketch one possible transformation in Fig. 4.

Given the restriction that a transition of N accesses at most one interface place, the translation from oWFN to service automata is straightforward and formalized through a mapping *oWFNtoService* from oWFN to service automata.

Definition 12 (Mapping oWFN to automata). *Let N be an oWFN where every transition accesses at most one interface place. Then $oWFNtoService(N)$ is the service automaton A with the following constituents:*

- *Q_A is the set of reachable markings of $inner(N)$;*
- *$I_A = P_i$, $O_A = P_o$;*
- *$[m, a, m'] \in \delta_A$ iff there is a transition t of N such that $[m, t, m']$ is a transition in the reachability graph of $inner(N)$ and either there is an interface place p connected to t and $a = p$, or t is not connected to any interface place and $a = \tau$;*
- *q_{0_A} is the initial state of $inner(N)$, F_A is the set of final states of $inner(N)$.*

The translation is justified through the following observation that can be easily verified by induction on the transition relations.

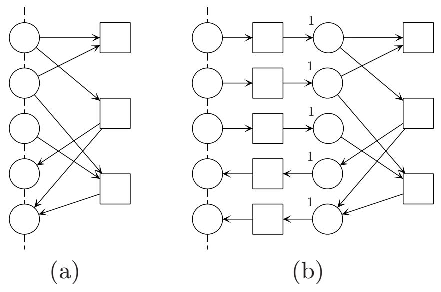


Fig. 4. In the oWFN (a), transitions are connected to several interface places. This can be circumvented by wrapping the interface with an additional internal buffer for each message channel that has capacity one (annotation to the places), cf. (b). This way, the essential behavior of the oWFN as well as finiteness of the state space are preserved.

Proposition 1. *For any two oWFN N and N' where every transition is connected to at most one interface place, the reachability graph of $N \oplus N'$ is isomorphic to the graph defined by the states and transitions of $oWFNtoService(N) \oplus oWFNtoService(N')$.*

In the remainder of this article, we study service automata *Prov* and *Req* where *Prov*, *Req*, and $Prov \oplus Req$ are all finite. This restriction implements the limited communication property introduced earlier. We return to oWFN in Sect. 6 where we discuss our implementation and report experimental results.

3 A Characterization of Deadlocks

In this section, we introduce concepts that help us to understand the coupling between two service automata *Prov* and *Req* from the point of view of *Req*. Therefore, we introduce the concept of *situations* which will allow us to characterize a deadlock in $Prov \oplus Req$ by considering *Req* only.

Definition 13 (K, Situation). *Let Prov and Req be partners. Then, let $K : Q_{Req} \rightarrow 2^{Q_{Prov} \times bags(C)}$ be defined by $K(q_{Req}) = \{[q_{Prov}, m] \mid [q_{Prov}, q_{Req}, m]$ is reachable from the initial state in $Q_{Prov \oplus Req}\}$. The elements of $2^{Q_{Prov} \times bags(C)}$ are called* situations.

A situation comprises all parts of a state of $Prov \oplus Req$ beyond the state of *Req* itself. It can thus be handled independently of *Req*. $K(q_{Req})$ can be interpreted as the *knowledge* that *Req* has about the possible states of *Prov* and the message bag, that is, the situations $[q_{Prov}, m]$ that can occur with q_{Req} in $Prov \oplus Req$.

We give some examples for values of K, referring to Fig. 2. We consider W as a partner of V. Then Fig. 3 tells us that $K(\mathsf{w0}) = \{[\mathsf{v0}, [\,]]\}$, $K(\mathsf{w1}) = \{[\mathsf{v0}, [\mathsf{e}]],$ $[\mathsf{v1}, [\,]]\}$, $K(\mathsf{w2}) = \{[\mathsf{v0}, [\mathsf{ce}]],$ $[\mathsf{v0}, [\mathsf{et}]],$ $[\mathsf{v1}, [\mathsf{c}]],$ $[\mathsf{v1}, [\mathsf{t}]],$ $[\mathsf{v2}, [\,]],$ $[\mathsf{v3}, [\,]],$ $[\mathsf{v0}, [\mathsf{C}]],$ $[\mathsf{v0}, [\mathsf{E}]],$ $[\mathsf{v0}, [\mathsf{T}]]\}$, $K(\mathsf{w3}) = \emptyset$, and $K(\mathsf{w4}) = \{[\mathsf{v0}, [\,]]\}$.

Within a set M of situations, we distinguish transient and stable situations. A situation is transient in M if a move of *Prov* in that situation leads to another situation also contained in M. Otherwise it is stable.

Definition 14 (Transient, stable situation). *Let M be a set of situations. $[q_{Prov}, m]$ is* transient *in M iff there is an $[q_{Prov}, x, q'_{Prov}] \in \delta_{Prov}$ such that:*
- *$x = \tau$ and $[q'_{Prov}, m] \in M$, or*
- *$x \in I_{Prov}$, $m(x) > 0$, and $[q'_{Prov}, m - [x]] \in M$, or*
- *$x \in O_{Prov}$ and $[q'_{Prov}, m + [x]] \in M$.*

Otherwise, $[q_{Prov}, m]$ is stable *in M.*

A service cannot leave a stable situation without interaction with the environment. For example, the situation $[\mathsf{v0}, [\mathsf{e}]]$ is transient in the set of situations $K(\mathsf{w1})$ (cf. Fig. 3). In contrast, the situation $[\mathsf{v1}, [\,]]$ is stable in $K(\mathsf{w1})$.

A deadlock in the composed system $Prov \oplus Req$ — seen from the point of view of *Req* only — now reads as follows.

Lemma 1. *$[q_{Prov}, q_{Req}, m]$ is a deadlock of $Prov \oplus Req$ if and only if all of the following conditions hold:*
- *$q_{Prov} \notin F_{Prov}$, or $q_{Req} \notin F_{Req}$, or $m \neq [\,]$;*
- *q_{Req} is a wait state of Req;*
- *$[q_{Prov}, m]$ is stable in $K(q_{Req})$ and $m(x) = 0$ for all $x \in wait(q_{Req})$.*

Proof. ($\rightarrow$) Let $[q_{Prov}, q_{Req}, m]$ be a deadlock. Then the first item is true by definition of deadlocks. The second item must be true since otherwise *Req* has a move. $[q_{Prov}, m]$ must be stable since otherwise *Prov* has a move. For $x \in wait(q_{Req})$, we can conclude $m(x) = 0$ since otherwise *Req* has a move.

($\leftarrow$) Assume, the three conditions hold. By the first item, the considered state is not a final state of $Prov \oplus Req$. *Prov* does not have a move since $[q_{Prov}, m]$ is stable. *Req* does not have a move since internal and send moves are excluded by the second item, and receive moves are excluded by the last item. □

Consider again the example deadlock $[\mathsf{v0}, \mathsf{x2}, [\mathsf{E}]]$ in $V \oplus X$ of the services of Fig. 2 and the three criteria of Lemma 1. Firstly, $[\mathsf{E}] \neq [\,]$. Secondly, $\mathsf{x2}$ is a wait state of X with $wait(\mathsf{x2}) = \{\mathsf{T}\}$. Thirdly, $K(\mathsf{x2}) = \{[\mathsf{v0}, [\mathsf{et}]], [\mathsf{v1}, [\mathsf{t}]], [\mathsf{v3}, [\,]], [\mathsf{v0}, [\mathsf{E}]], [\mathsf{v0}, [\mathsf{T}]]\}$ and $[\mathsf{v0}, [\mathsf{E}]]$ is stable in $K(\mathsf{x2})$ and $[\mathsf{E}](\mathrm{T}) = 0$. Hence, all criteria hold and we can conclude that $[\mathsf{v0}, \mathsf{x2}, [\mathsf{E}]]$ is indeed a deadlock.

For a state $[q_{Prov}, q_{Req}, m]$, the three requirements of Lemma 1 can be easily compiled into Boolean formulas $\phi_1(q_{Prov}, m)$, ϕ_2, and $\phi_3(m)$ which express the absence of deadlocks of the shape $[\cdot, q_{Req}, \cdot]$ in $Prov \oplus Req$. The formulas use the set of propositions $C \cup \{\tau, \mathit{final}\}$ (with $\mathit{final} \notin C$). Propositions in $C \cup \{\tau\}$ represent labels of transitions that leave q_{Req}, whereas proposition *final* represents the fact whether $q_{Req} \in F_{Req}$:

Definition 15 (Annotation, *Req*-assignment). *Let Prov and Req be partners. Then, for each $q_{Req} \in Q_{Req}$, define the* annotation *$\phi(q_{Req})$ of q_{Req} as the Boolean formula over the propositions $C \cup \{\tau, \mathit{final}\}$ as follows.*

$$\phi(q_{Req}) = \bigwedge\nolimits_{[q_{Prov}, m] \text{ stable in } K(q_{Req})} (\phi_1(q_{Prov}, m) \vee \phi_2 \vee \phi_3(m))$$

where

- $\phi_1(q_{Prov}, m) = \begin{cases} \mathit{final}, & \text{if } q_{Prov} \in F_{Prov} \text{ and } m = [\,], \\ \mathit{false}, & \text{otherwise,} \end{cases}$
- $\phi_2 = \tau \vee \bigvee_{x \in O_{Req}} x,$
- $\phi_3(m) = \bigvee_{x \in I_{Req}, m(x) > 0} x.$

The Req-assignment $ass_{Req}(q_{Req}) : C \cup \{\tau, \mathit{final}\} \to \{\mathit{true}, \mathit{false}\}$ *assigns true to a proposition* $x \in C \cup \{\tau\}$ *iff there is a* q'_{Req} *such that* $[q_{Req}, x, q'_{Req}] \in \delta_{Req}$ *and true to the proposition final iff* $q_{Req} \in F_{Req}$.

Since the formula $\phi(q_{Req})$ exactly reflects Lemma 1, we obtain:

Corollary 1. *Prov* $\oplus$ *Req is deadlock-free if and only if, for all* $q_{Req} \in Q_{Req}$, *the value of* $\phi(q_{Req})$ *with the Req-assignment* $ass_{Req}(q_{Req})$ *is true.*

In Fig. 2, the annotation of state w1 of W would be $\tau \vee \mathsf{e} \vee \mathsf{c} \vee \mathsf{t} \vee \mathsf{q}$, due to the single stable situation $[\mathsf{v1}, [\,]] \in K(\mathsf{w1})$. This formula is satisfied by the W-assignment that assigns *true* to both c and t, and *false* to τ, e, and q in state w1. The annotation of the state w2 is $\tau \vee \mathsf{e} \vee \mathsf{c} \vee \mathsf{t} \vee \mathsf{q} \vee (\mathsf{C} \wedge \mathsf{E} \wedge \mathsf{T})$ since $K(\mathsf{w2})$ contains the three stable situations $[\mathsf{v0}, [\mathsf{C}]]$, $[\mathsf{v0}, [\mathsf{E}]]$, and $[\mathsf{v0}, [\mathsf{T}]]$. Since the W-assignment assigns *true* to all of C, E, and T in state w2, it satisfies the annotation. For state x2 of X, the annotation is $\tau \vee \mathsf{e} \vee \mathsf{c} \vee \mathsf{t} \vee \mathsf{q} \vee (\mathsf{T} \wedge \mathsf{E})$. Since the only transition leaving x2 is T, the X-assignment assigns *false* to all propositions except T in state x2, and the annotation yields *false*. This corresponds to the deadlock $[\mathsf{v0}, \mathsf{x2}, [\mathsf{E}]]$ in $V \oplus X$.

4 A Canonical Partner

We are now ready to compute a canonical service automaton, called S, which interacts properly with a given service *Prov*.

For any finite k-limited communication partner of a given (finite) service automaton *Prov*, all reachable situations are actually in $2^{Q_{Prov} \times bags_k(C)}$ which is a finite domain. For sets of situations, define the following operations.

Definition 16 (Closure). *For a set M of situations, let the* closure *of* M, *denoted* $cl(M)$, *be inductively defined as follows.*
Base: $M \subseteq cl(M)$;
Step: If $[q_{Prov}, m] \in cl(M)$ *and* $[q_{Prov}, x, q'_{Prov}] \in \delta_{Prov}$, *then*
- $[q'_{Prov}, m] \in cl(M)$, *if* $x = \tau$,
- $[q'_{Prov}, m + [x]] \in cl(M)$, *if* $x \in O_{Prov}$,
- $[q'_{Prov}, m - [x]] \in cl(M)$, *if* $x \in I_{Prov}$ *and* $m(x) > 0$.

It can be easily seen that $cl(M)$ comprises those situations that can be reached from a situation in M without interference from a partner. In Fig. 2 for example, we obtain $cl(\{[\mathsf{v0}, [\mathsf{ce}]]\}) = \{[\mathsf{v0}, [\mathsf{ce}]], [\mathsf{v1}, [\mathsf{c}]], [\mathsf{v2}, [\,]], [\mathsf{v0}, [\mathsf{C}]], [\mathsf{v0}, [\mathsf{E}]]\}$.

Definition 17 (Send-event, receive-event, internal-event) *Let $M \subseteq Q_{Prov} \times bags(C)$. If $x \in O_{Prov}$, then the* send-event *x, $send(M,x)$, is defined as $send(M,x) = \{[q, m + [x]] \mid [q,m] \in M\}$. If $x \in I_{Prov}$, then the* receive-event *x, $receive(M,x)$, is defined as $receive(M,x) = \{[q, m - [x]] \mid [q,m] \in M, m(x) > 0\}$. The* internal-event *$\tau$, $internal(M,\tau)$, is defined as $internal(M,\tau) = M$. As the shape of an event is clear from I_{Prov} and O_{Prov}, we define the* event *x, $event(M,x)$, as $receive(M,x)$ if $x \in I_{Prov}$, $send(M,x)$ if $x \in O_{Prov}$, and $internal(M,x)$ if $x = \tau$.*

A send-event models the effect that a message sent by *Req* has on a set of situations M. A receive-event models the effect that a message received by *Req* has on a set of situations M. Considering the service V of Fig. 2, we get, for example, $receive(\{[\mathsf{v0},[\mathsf{ce}]],[\mathsf{v1},[\mathsf{c}]],[\mathsf{v2},[\,]],[\mathsf{v0},[\mathsf{C}]],[\mathsf{v0},[\mathsf{E}]]\},\mathsf{C}) = \{[\mathsf{v0},[\,]]\}$ and $send(\{[\mathsf{v0},[\mathsf{e}]],[\mathsf{v1},[\,]]\},\mathsf{c}) = \{[\mathsf{v0},[\mathsf{ce}]],[\mathsf{v1},\mathsf{c}]]\}$.

Now, the construction of S (Def. 18) bases on the following considerations. A *state* of S is a *set of situations*. States and transitions are organized such that, for all states q of S, $K(q) = q$, that is, every state of S is equal to the set of situations it can occur with in the composition with *Prov*. The transitions of S can be determined using the operations *event* and *cl*. The construction is restricted to sets in $2^{Q_{Prov} \times bags_k(C)}$. With this restriction, we already implement the property of k-limited communication. Given the desired property $K(q) = q$ and the definition of K, the composed system cannot enter a state violating k-limited communication. The other way round, any reachable state q where $K(q)$ is outside $2^{Q_{Prov} \times bags_k(C)}$ would cause a violation of that property.

Starting with a service automaton S_0 which contains *all* such states and transitions, unfortunately, $S_0 \oplus Prov$ may contain deadlocks. However, these deadlocks can be identified by annotating S_0 and evaluating the annotations according to Def. 15. Removing all states where the annotation evaluates to *false* yields a new structure S_1. Since it is possible that the initial state is among the removed ones, S_1 is not necessarily a service automaton, that is, it is not well-defined. In that case, however, we are able to prove that *Prov* does not have correctly interacting partners at all. By removing states, assignments of remaining states can change their values. Thus, the removal procedure is iterated until either the initial state is removed, or all annotations eventually evaluate to *true*. In the latter case, the remaining service automaton is called S and, by construction of the annotations, constitutes a partner that interacts properly with *Prov*.

Definition 18 (Canonical partner S). *Let Prov be a service automaton and assume a number k to be given. Define inductively a sequence of (not necessarily well-defined) service automata $S_i = [Q_i, I_i, O_i, \delta_i, q_{0i}, F_i]$ as follows.*

Let $Q_0 = 2^{Q_{Prov} \times bags_k(C)}$. Let, for all i, $I_i = O_{Prov}$, $O_i = I_{Prov}$, $q_{0i} = cl(\{[q_{0Prov}, [\,]]\})$, $[q, x, q'] \in \delta_i$ iff $q, q' \in Q_i$ and $q' = cl(event(q,x))$, and $F_i = \{q \in Q_i \mid q$ is wait state of $S_i\}$. Let, for all i, $Q_{i+1} = \{q \mid q \in Q_i, \phi(q)$ evaluates to true with assignment $ass_{S_i}(q)\}$.

Let S be equal to S_i for the smallest i satisfying $S_i = S_{i+1}$.

As the sequence $\{S_i\}_{i=0,1,...}$ is monotonously decreasing, all objects of this definition are uniquely defined. The resulting S is a well-defined service automaton if and only if $q_{0_S} \in Q_S$. In that case, S is in fact a k-limited deadlock-freely interacting partner of *Prov*.

As an example, the partner service S_0 for the service V of Fig. 2 initially consists of 21 (from $q_{0_{S_0}}$ reachable) states from which 9 states are removed during the computation of the canonical partner S for V. The resulting service automaton S can be found as the underlying graph of Fig. 7 in Sect. 5.

With the next few results, we further justify the construction.

Lemma 2. *If $cl([q_{0Prov}, [\,]]) \not\subseteq Q_{Prov} \times bags_k(C)$, then Prov does not have k-limited communication partners for the number k used in the construction.*

Proof. As $cl([q_{0Prov}, [\,]])$ is the set of situations that can be reached from the initial state without interference of any partner *Req*, k-limited communication is immediately violated. □

Lemma 2 states that S_0 is well-defined for all well-designed *Prov*, that is, for all *Prov* such that there exists at least one partner *Req* with $Prov \oplus Req$ is deadlock-free.

The next lemma shows that we actually achieved one of the major goals of the construction.

Lemma 3. *For all S_i and all $q \in Q_i$: if q is δ_i-reachable from q_{0_i}, then $K(q) = q$.*

Proof. By structural induction on δ. By definition of *cl*, $cl([q_{0Prov}, [\,]])$ is the set of situations that can be reached from the initial state without interference from S_i. If $K(q) = q$, then $cl(event(q, x))$ is by definition of *event* and *cl* exactly the set of situations that can be reached from situations in q by the event x. Thus, $[q, x, q'] \in \delta_i$ implies $K(q') = q'$. □

From that lemma we can directly conclude that the service S constitutes a properly interacting partner of *Prov*.

Corollary 2. *If S is well-defined, that is, $q_{0_S} \in Q_S$, then $Prov \oplus S$ is deadlock-free.*

Proof. Follows with Lemma 1 and Def. 15 from the fact that all states of S satisfy their annotations. □

As an example of an ill-designed service, the service Z in Fig. 5(a) would yield an infinite $cl([\mathsf{z0}, [\,]])$ for any partner. Accordingly, there is no well-defined S_0. During the construction of S for service U in Fig. 5(b), the initial state is eventually removed. The initial state $\{[\mathsf{u0}, [\,]], [\mathsf{u1}, [\,]], [\mathsf{u2}, [\,]]\}$ of S_0 for U has two successors. The a-successor $\{[\mathsf{u0}, [\mathsf{a}]], [\mathsf{u1}, [\mathsf{a}]], [\mathsf{u2}, [\mathsf{a}]], [\mathsf{u3}, [\,]]\}$ must be removed since it contains the deadlock $[\mathsf{u2}, [\mathsf{a}]]$, the b-successor must be removed since it contains the deadlock $[\mathsf{u1}, [\mathsf{b}]]$. In the next iteration, the initial state itself must be removed since, without the two successors, it violates its annotation $(\mathsf{a} \wedge \mathsf{b}) \vee \tau$.

For further studying the constructed partner S, we establish a matching relation between states of services and apply this relation to relate states of an arbitrary partner *Req* of *Prov* to states of the canonical partner S.

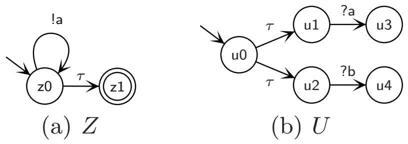


Fig. 5. The service Z has no k-limited communication partner (for any number k). The service U cannot communicate deadlock-freely with any partner.

Definition 19 (Matching). *Let A and B be service automata and define the relation $L_{A,B} \subseteq Q_A \times Q_B$, the* matching *between A and B, inductively as follows. Let $[q_{0_A}, q_{0_B}] \in L_{A,B}$. If $[q_A, q_B] \in L_{A,B}$, $[q_A, x, q'_A] \in \delta_A$ and $[q_B, x, q'_B] \in \delta_B$, then $[q'_A, q'_B] \in L_{A,B}$.*

The matching between two services A and B is a strong simulation relation where in particular one τ-step of A is related to exactly one τ-step of B.

Examples for matchings are shown in Fig. 6. For example, the state w2 of the service W in Fig. 6(a) is matched with the states 4 and 6 of the service S in Fig. 7.

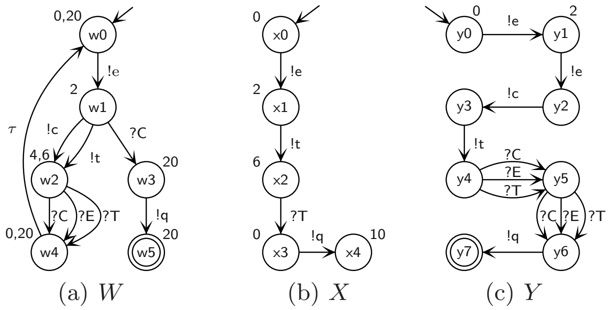


Fig. 6. Matching of the three services W, X, and Y of Fig. 2 with the service S depicted in Fig. 7. A number n attached to a state x represents a pair $[x, n] \in L$.

Lemma 4. *Let S_0 be the starting point of the construction in Def. 18 and Req be a partner of Prov with k-limited communication (for the value of k used in the construction above). For all $q_{Req} \in Q_{Req}$, $K(q_{Req}) = \bigcup_{[q_{Req}, q_{S_0}] \in L_{Req,S_0}} K(q_{S_0})$.*

Proof (Sketch). The inclusion $K(q_{Req}) \supseteq \bigcup_{[q_{Req}, q_{S_0}] \in L_{Req,S_0}} q_{S_0}$ follows from the definition of q_{0S_0}, δ_{S_0}, and the concepts *cl* and *event*. For the reverse inclusion, let $[q_{Prov}, m] \in K_{q_{Req}}$, that is, $[q_{Prov}, q_{Req}, m] \in Prov \oplus Req$. Thus, there is a transition sequence in $Prov \oplus Req$ from the initial state $[q_{0Prov}, q_{0Req}, [\,]]$ to that state. This sequence can be replayed in $Prov \oplus S_0$ by replacing actions of Req with actions of S_0, leading to a state q_{S_0} with $[q_{Prov}, m] \in K(q_{S_0}) = q_{S_0}$. □

Corollary 3. *For each state q_{Req} of Req, its annotation $\phi(q_{Req})$ can be described as $\phi(q_{Req}) \equiv \bigwedge_{q_{S_0}:[q_{Req},q_{S_0}]\in L_{Req,S_0}} \phi(q_{S_0})$.*

Proof. Since an annotation $\phi(q_{S_0})$ is a conjunction built for every situation in $q_{S_0} = K(q_{S_0})$, the annotation of the union of K-values is the conjunction of the individual formulas. $\square$

For example, the annotation of state w2 of the service in Fig. 2 is $(\mathsf{C} \wedge \mathsf{E} \wedge \mathsf{T}) \vee \mathsf{c} \vee \mathsf{e} \vee \mathsf{t} \vee \tau$ which is equivalent to the conjunction of the annotations $(\mathsf{C} \wedge \mathsf{E}) \vee \tau$ and $(\mathsf{E} \wedge \mathsf{T}) \vee \mathsf{c} \vee \mathsf{e} \vee \mathsf{t} \vee \tau$ of the states 4 and 6 of the service in Fig. 7.

The next result is the actual justification of the removal process described in Def. 18.

Lemma 5. *If Req is a k-limited communication partner of Prov (for the value of k used in the construction of S_0) such that $Prov \oplus Req$ is deadlock-free, then $q_S \in S$ for all $[q_{Req}, q_S] \in L_{Req,S_0}$.*

Proof. (By contradiction) Let i be the smallest number such that there exist $q_{Req} \in Q_{Req}$ and $q_S \in Q_{S_i} \setminus Q_{S_{i+1}}$ holding $[q_{Req}, q_S] \in L$. That is, q_S is, among the states of S_0 appearing in L_{Req,S_0}, the one that is removed first during the process described in Def. 18.

By the construction of Def. 18, $\phi(q_S)$ evaluates to *false* with the assignment $ass_{S_i}(q_S)$. Thus, there is a $[q_{Prov}, m] \in K(q_S)$ such that $[q_{Prov}, q_S, m]$ is a deadlock in $Prov \oplus S_i$. As a deadlock, it is also a wait state in S_i, so $q_S \in F_{S_i}$.

In S_0, there is, for every $x \in C \cup \{\tau\}$, a transition leaving q_S. If such a transition is not present from q_S in S_i, this means that the corresponding successor state has been removed in an earlier iteration of the process described in Def. 18. Such a transition cannot leave q_{Req} in Req since otherwise a successor of Req were matched with a state q'_S that has been removed in an earlier iteration than q_S which contradicts the choice of i and q_S. Consequently, for every x with an x-transition leaving q_{Req} in Req, there is an x-transition leaving q_S in S_i. This means that, for all $x \in C \cup \{\tau, \mathit{final}\}$, $ass_{S_i}(q_S)(x) \geq ass_{Req}(q_{Req})(x)$. Since $\phi(q_S)$ is monotonous (only built using $\vee$ and $\wedge$), and ϕ_{Req} is a conjunction containing $\phi(q_S)$ (by. Cor. 3), $\phi(q_{Req})$ evaluates to *false* with the assignment $ass_{Req}(q_{Req})$. Consequently, by Cor. 1, $Prov \oplus Req$ has a deadlock. $\square$

Corollary 4. *Prov has a k-limited communication partner Req (for the value of k used in the construction of S_0) such that $Prov \oplus Req$ is deadlock-free if and only if $q_{0S} \in Q_S$ (i. e. the service-automaton S is well-defined).*

Proof. If S is well-defined then, by Cor. 2, at least S is a partner of *Prov* such that $Prov \oplus S$ is deadlock-free. If *Prov* has a partner *Req* such that $Prov \oplus Req$ is deadlock-free, Lemma 5 asserts that L_{Req,S_0} contains only states of S. In particular, since in any case $[q_{0Req}, q_{0S_0}] \in L_{Req,S_0}$, this implies $q_{0S_0} = q_{0S} \in Q_S$. $\square$

If *Prov* does not have partners *Req* such that $Prov \oplus Req$ is deadlock-free, then *Prov* is fundamentally ill-designed. Otherwise, the particular partner S studied above is well-defined. It is the basis for the concept of an operating guideline for *Prov* which is introduced in the next section.

5 Operating Guidelines

If the matching of a service *Req* with S_0 involves states of $Q_{S_0} \setminus Q_S$, Lemma 5 asserts that $Prov \oplus Req$ has deadlocks. In the case that the matching involves only states of Q_S, $Prov \oplus Req$ may or may not have deadlocks. However, by Cor. 1, the existence of deadlocks in $Prov \oplus Req$ can be decided by evaluating the annotations $\phi(q_{Req})$ for the states $q_{Req} \in Q_{Req}$. By Cor. 3, these formulas can be retrieved from the annotations to the states of S. Attaching these formulas explicitly to the states of S, the whole process of matching and constructing the $\phi(q_{Req})$ can be executed without knowing the actual contents of the states of S, that is, without knowing the situations — the topology of S is sufficient. This observation leads us to the concept of an operating guideline for *Prov*.

Definition 20 (Operating guideline). *Let Prov be a service automaton which has at least one properly interacting partner and S be the canonical service automaton of Def. 18. Then any automaton S^* that is isomorphic to S under isomorphism h, together with an annotation Φ with $\Phi(q_{S^*}) = \phi(h(q_{S^*}))$ for all q_{S^*}, is called* operating guideline for *Prov.*

With the step from S to an isomorphic S^*, we just want to emphasize that only the topology of S is relevant in the operating guideline while the internal structure of the states of S, that is, the set of situations, is irrelevant.

Figure 7 shows the operating guideline, that is, the annotated service automaton S, for the service V of Fig. 2.

Ignoring the annotations, the operating guideline is (isomorphic to) the partner S for *Prov* that can be executed directly thus satisfying the second requirement stated in the introduction. The annotation at a state q gives additional instructions about whether or not transitions leaving q may be skipped. Therefore, the operating guideline can be used to decide, for an arbitrary service *Req*, whether or not $Prov \oplus Req$ is deadlock-free, as the next result shows.

Theorem 1 (Main theorem of this article). *Let Prov be a finite state service and S^* its operating guideline. Req is a k-limited communication partner of Prov (for the value of k used in the construction of S^*) such that $Prov \oplus Req$ is deadlock-free if and only if the following requirements hold for every $[q_{Req}, q_{S^*}] \in L_{Req,S^*}$:*

(topology) *For every $x \in C \cup \{\tau\}$, if there is an x-transition leaving q_{Req} in Req, then there is an x-transition leaving q_{S^*} in S^*.*
(annotation) *The assignment $ass_{Req}(q_{Req})$ satisfies $\Phi(q_{S^*})$.*

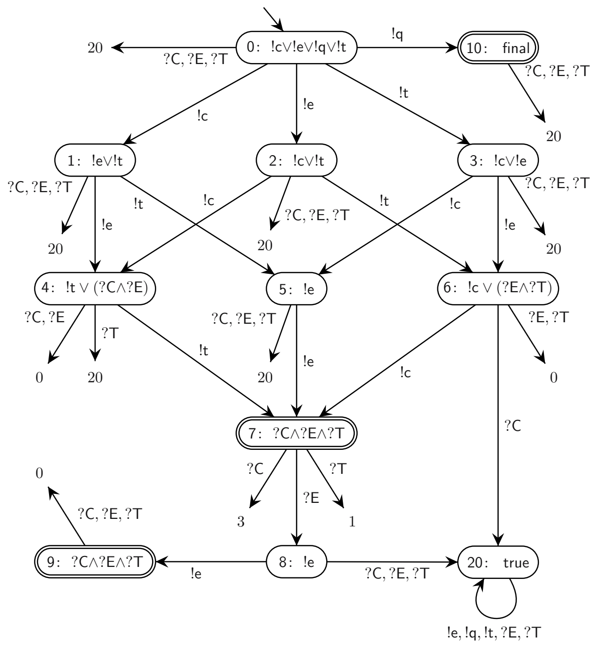


Fig. 7. The operating guideline for the service V in Fig. 2. It is isomorphic to the canonical partner S for V (Def. 18) with the annotations Φ depicted inside the nodes. Multiple labels to an edge mean multiple edges. Edges pointing to a number mean edges to the node with the corresponding number.

Note that this theorem matches *Req* with S^* (isomorphic to S) while the results in the previous section match *Req* with S_0. Requirement (topology) actually states that L_{Req,S^*} is a simulation relation.

Proof. If $Prov \oplus Req$ is deadlock-free, then Lemma 5 asserts that the matching of *Req* with S (or S^*) coincides with the matching of *Req* with S_0. Thus, requirement (topology) holds. Furthermore, Cor. 3 guarantees that requirement (annotation) is satisfied.

Assume that both requirements hold. By requirement (topology), the matching of *Req* with S (or S^*) coincides with the matching of *Req* with S_0, since the matching with S_0 can lead to states outside S only if there is an x such that an x-transition is present in a state q_{Req} but not in the corresponding state $q_S \in S$. Given that both matchings coincide, Cor. 3 states that $\phi(q_{Req})$ is the conjunction of the $\phi(q_S)$, for the matching states q_S. Then, we can deduce from Cor. 1 and requirement (annotation) that $Prov \oplus Req$ is deadlock-free. □

Consider the service V of Fig. 2 and its partners. In Fig. 6 we can see that W and X satisfy the requirement (topology) while Y does not (Y is not a 1-limited communication partner of V). X violates in state x2 the annotation to the matched state 6, since the *Req*-assignment in state x2 assigns *false* to E and

τ. $V \oplus X$ contains the deadlock $[\mathsf{v0}, \mathsf{x2}, [\mathsf{E}]]$. For service W, all annotations are satisfied. $V \oplus W$ is deadlock-free (see Fig. 3).

At this stage, it would be natural to ask for an operating guideline that has the shape of an oWFN rather than an automaton. While the partner S can be transformed to an oWFN using existing approaches called region theory [18], we have no concept of transforming the annotations of the states of S into corresponding annotations of the resulting oWFN. That is why our concept of operating guidelines is presented on the level of service automata.

6 Implementation

All concepts used in this article have been defined constructively. For an actual implementation, it is, however, useful to add some ideas that increase efficiency. First, it is easy to see that, for constructing S, it is not necessary to start with the whole S_0. For the matching, only states that are reachable from the initial state need to be considered. Furthermore, annotations can be generated as soon as a state is calculated. They can be evaluated as soon as the immediate successors have been encountered. If the annotation evaluates to *false*, further exploration can be stopped [19]. In consequence, the process of generating S_0 can be interleaved with the process of removing states that finally leads to S. This way, memory consumption can be kept within reasonable bounds.

The number of situations in a state q of S can be reduced using, for instance, partial order reduction techniques. In ongoing research, we explore that possibility. We are further exploring opportunities for a compact representation of an operating guideline. For this purpose, we already developed a *binary decision diagram* (BDD, [20]) representation of an operating guideline for acyclic services that can be efficiently used for matching [21]. Most likely, these concepts can be adapted to arbitrary finite-state services.

We prototypically implemented our approach within the tool Fiona [10]. Among other features, Fiona can read an open workflow net and generate the operating guideline. The following example Petri nets stem from example WS-BPEL specifications of services. The WS-BPEL processes have been translated automatically into oWFN, based on the Petri net semantics for WS-BPEL [9] and the tool BPEL2oWFN [10].

The "Purchase Order" and "Loan Approval" processes are realistic services taken from the WS-BPEL specification [8]. "Olive Oil Ordering" [22], "Help Desk Service Request" (from the Oracle BPEL Process Manager) and "Travel Service" [8] are other web services that use WS-BPEL features like fault and event handling. The "Database Service" shows that it may be necessary to calculate a number of situations which is a multiple of the number of states of the considered service automaton. "Identity Card Issue" and "Registration Office" are models of administrative workflows provided by Gedilan, a German consulting company. Finally, we modeled parts of the Simple Mail Transfer Protocol (SMTP) [23]. Since it is a communication protocol, it yields the biggest operating guideline.

Table 1. Experimental results running Fiona. All experiments were taken on a Intel Pentium M processor with 1.6 GHz and 1 GB RAM running Windows XP.

	open workflow net				inner	operating guideline			time
service *Prov*	places	input	output	trans.	states	situations	states	edges	(sec)
Purchase Order	38	4	6	23	90	464	168	548	0
Loan Approval	48	3	3	35	50	199	7	8	0
Olive Oil Ordering	21	3	3	15	15	5101	14	20	0
Help Desk Service	33	4	4	28	25	7765	8	10	2
Travel Service	517	6	7	534	1879	5696	320	1120	7
Database Service	871	2	5	851	5232	337040	54	147	7583
Identity Card Issue	149	2	9	114	111842	707396	280	1028	216
Registration Office	187	3	3	148	7265	9049	7	8	0
SMTP	206	8	4	215	7111	304284	362	1161	200

Table 1 provides the size of the open workflow net and the number of states of the corresponding service automaton (i. e., the inner of the oWFN), the size (number of situations, states, and edges) of the calculated operating guideline, and the time for its calculation from the given Petri net.

The examples show that operating guidelines for realistic services have reasonable size. Considering the still unexplored capabilities of reduction techniques and symbolic representations, we may conclude that the operating guideline approach is in fact feasible for tasks like service discovery (where it needs to be complemented with mechanisms for matching semantic issues).

7 Conclusion

With the concept of an operating guideline for a service *Prov*, we proposed an artifact that can be directly executed. The operating guideline is expressed in terms of the interface of *Prov*, and gives complete information about correct communication with *Prov*. It can be manipulated in accordance with the annotations. This way, other partners can be crafted which, by construction, communicate correctly with *Prov*, too. These partners can be translated into other formalisms, most notably, oWFN.

Deciding correct interaction using an operating guideline amounts to checking the simulation relation between the partner service and the operating guideline and evaluating the annotations. It has about the same complexity as model checking deadlock freedom in the composed system itself. Due to its completeness, and due to its explicit operational structure, it can be a valuable tool in service-oriented architectures.

Experimental results have shown that the calculation of an operating guideline is feasible in practical applications.

In future work, we want to adapt the operation guideline concept to infinite-state services. However, we have strong evidence that, given an infinite-state service, the problem to construct a deadlock-freely interacting partner service is undecidable in this scenario. Besides, we want to study whether it is possible to construct operating guidelines for services without k-limited communication partners. Finally, we also want to characterize livelock-free interactions. As first examples show, it is not trivial to cover livelock-freedom by Boolean annotations.

References

1. Massuthe, P., Schmidt, K.: Operating guidelines – An automata-theoretic foundation for the service-oriented architecture. In: QSIC 2005, pp. 452–457. IEEE Computer Society Press, Washington (2005)
2. Massuthe, P., Reisig, W., Schmidt, K.: An Operating guideline approach to the SOA. Annals of Mathematics, Computing and Teleinformatics 1(3), 35–43 (2005)
3. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-services: A look behind the curtain. In: PODS '03, pp. 1–14. ACM Press, New York (2003)
4. Papazoglou, M.P.: Agent-oriented technology in support of e-business. Communications of the ACM 44(4), 71–77 (2001)
5. Aalst, W.v.d., Weske, M.: The P2P approach to interorganizational workflows. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 140–156. Springer, Berlin (2001)
6. Leymann, F., Roller, D., Schmidt, M.: Web services and business process management. IBM Systems Journal, vol. 41(2) (2002)
7. Aalst, W.v.d.: The application of petri nets to workflow management. Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
8. Alves, A. et al.: Web Services Business Process Execution Language Version 2.0. Committee Draft, 25 January, 2007, OASIS (2007)
9. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 220–235. Springer, Heidelberg (2005)
10. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In: Dustdar, S., Fiadeiro, J.L., Sheth, A. (eds.) BPM 2006. LNCS, vol. 4102, pp. 17–32. Springer, Heidelberg (2006)
11. Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and Validation of the Business Process Execution Language for Web Services. In: Zimmermann, W., Thalheim, B. (eds.) ASM 2004. LNCS, vol. 3052, pp. 78–94. Springer, Heidelberg (2004)
12. Fahland, D., Reisig, W.: ASM-based semantics for BPEL: The negative Control Flow. In: Proceedings of the 12th International Workshop on Abstract State Machines (ASM'05), Paris XII, pp. 131–151 (2005)
13. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
14. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic composition of e-services that export their behavior. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 43–58. Springer, Heidelberg (2003)
15. Arias-Fisteus, J., Fernández, L.S., Kloos, C.D.: Formal Verification of BPEL4WS Business Collaborations. In: Bauknecht, K., Bichler, M., Pröll, B. (eds.) EC-Web 2004. LNCS, vol. 3182, pp. 76–85. Springer, Heidelberg (2004)
16. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, pp. 621–630. ACM Press, New York (2004)

17. Ferrara, A.: Web services: a process algebra approach. In: ICSOC, ACM 2004, pp. 242–251, ACM (2004)
18. Badouel, E., Darondeau, P.: Theory of regions. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
19. Weinberg, D.: Reduction Rules for Interaction Graphs. Techn. Report 198, Humboldt-Universität zu Berlin (2006)
20. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
21. Kaschner, K., Massuthe, P., Wolf, K.: Symbolische Repräsentation von Bedienungsanleitungen für Services. In: AWPN Workshop 2006, Universität Hamburg (in German) pp. 54–61(2006)
22. Arias-Fisteus, J., Fernández, L.S., Kloos, C.D.: Applying model checking to BPEL4WS business collaborations. In: Proceedings of the, ACM Symposium on Applied Computing (SAC), pp. 826–830, ACM (2005)
23. Postel, J.B.: Simple Mail Transfer Protocol. RFC 821, Information Sciences Institute, University of Southern California, Network Working Group (1982)

Theory of Regions for the Synthesis of Inhibitor Nets from Scenarios

Robert Lorenz, Sebastian Mauser, and Robin Bergenthum

Department of Applied Computer Science,
Catholic University of Eichstätt-Ingolstadt
firstname.lastname@ku-eichstaett.de

Abstract. In this paper we develop a theory for the region-based synthesis of system models given as place/transition-nets with weighted inhibitor arcs (pti-nets) from sets of scenarios describing the non-sequential behaviour. Scenarios are modelled through labelled stratified order structures (LSOs) considering "earlier than" and "not later than" relations between events [6,8] in such a way that concurrency is truly represented.

The presented approach generalizes the theory of regions we developed in [10] for the synthesis of place/transition-nets from sets of labelled partial orders (LPOs) (which only model an "earlier than" relation between events). Thereupon concrete synthesis algorithms can be developed.

1 Introduction

Synthesis of Petri nets from behavioural descriptions has been a successful line of research since the 1990ies. There is a rich body of nontrivial theoretical results and there are important applications in industry, in particular in hardware design [3], in control of manufacturing systems [15] and recently also in workflow design [13,14].

The synthesis problem is the problem to construct, for a given behavioural specification, a Petri net of a considered Petri net class such that the behaviour of this net coincides with the specified behaviour (if such a net exists). There exist theories for the synthesis of place/transition-nets (p/t-nets) from behavioural models describing sequential semantics [1], step semantics [1] and partial order semantics [10]. There are also sequential, respectively step semantics, based approaches for the synthesis of elementary nets [4,5] and extensions to elementary nets with inhibitor arcs [2,11,12].

In this paper we generalize the synthesis theory for partial order semantics from [10] to p/t-nets with weighted inhibitor arcs (pti-nets). In [10] the behavioural specification is given by a set of labelled partial orders (LPOs) – a so called partial language – interpreted as a scenario-based description of the non-sequential behaviour of p/t-nets. The aim in [10] is the characterization and synthesis of a p/t-net whose behaviour coincides with a given partial language. That means, the LPOs of the partial language should exactly be the partially ordered executions of the searched p/t-net. Note hereby that partial languages regard the most general concurrency relationships between events (in contrast to sequential semantics considering no concurrency relations and step semantics considering only restricted transitive concurrency relations).

The synthesis of the p/t-net is based on the notion of regions: The p/t-net synthesized from a partial language inherits its transitions from the event labels of the LPOs which in turn describe the respective occurring actions. Through places causal dependencies between transitions are added restricting the set of executions. The idea is to add all places which do not restrict the set of executions too much in the sense that they do not prohibit the executability of any LPO specified in the partial language. These places are called feasible (w.r.t. the given partial language). Adding all feasible places yields a p/t-net – the so called saturated feasible p/t-net – which has a minimal set of partially ordered executions including the specified partial language (among all p/t-nets). Consequently the saturated feasible p/t-net solves the synthesis problem or there exits no solution of the problem. The general approach of a theory of regions is to determine feasible places by so called regions of the behavioural model.[1] As the main result in [10] we proposed a notion of regions for partial languages and showed that the set of regions exactly defines the set of feasible places. In this paper we lift this approach to the level of pti-nets. That means we generalize the notion of regions to a scenario-based behavioural model of pti-nets and show that these regions exactly define feasible places.

In the following we introduce the scenario-based behavioural model of pti-nets considered in this paper. We will examine the so called a-priori semantics of pti-nets [8] in which synchronicity of events is explicitly regarded.[2] Thus, as the model of non-sequential behaviour we consider a generalization of LPOs – so called labelled stratified order structures (labelled so-structures or LSOs) [6,8].[3] That means, given a pti-net, scenarios are specified by LSOs with transition names as event labels, and a specified scenario may be or may not be an execution of the net.

In an LPO ordered events are interpreted as causally dependent in the sense of an ”earlier than” relation. Unordered events are considered as causally independent respectively concurrent. That means two events are concurrent, if they can occur in arbitrary order as well as synchronously. Thus, synchronicity cannot be distinguished from concurrency in the case of LPOs. A situation (1.) in which two events a and b can only occur synchronously or (2.) can occur synchronously and in the order $a \to b$, but not in the order $b \to a$, cannot be modelled with LPOs (obviously in both situations (1.) and (2.) the events are not concurrent, but synchronous occurrence is possible). For these situations LSOs include a ”not later than” relation between events: a ”not later than” b exactly describes (2.) and a symmetric ”not later than” relation between events (a ”not later than” b and b ”not later than” a) models (1.). Thus, an LSO is based on an LPO (the ”earlier than” relation is depicted with solid arcs in illustrations), to which a ”not later than” relation (dashed arcs) between events is consistently added.

In [6] it was explained in detail that the ”earlier than” relation of LPOs is not enough to describe executions of some Petri net classes such as inhibitor nets under the a-priori semantics and that LSOs form the adequate behavioural model for these net classes. In Figure 1 this phenomenon is illustrated: A pti-net and four LSOs describing executions

[1] For sequential or step semantics this theory lead to polynomial synthesis algorithms [1].

[2] There are also alternative semantics of inhibitor nets. The a-posteriori semantics (which is less general than the a-priori semantics from a causal point of view) is discussed in the conclusion.

[3] Note that just like LPOs in the case of p/t-nets, LSOs can model arbitrary dependency relations between transition occurrences of pti-nets, i.e. concurrency can be truly represented.

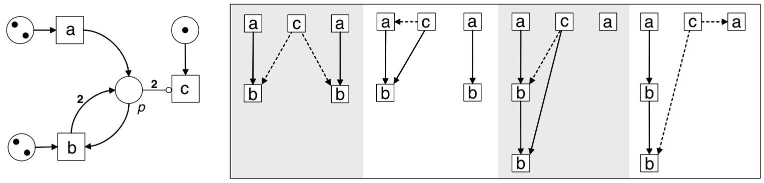


Fig. 1. A pti-net together with some executions

of the net are depicted. The pti-net has the only inhibitor arc (p, c) with inhibitor weight two. This arc restricts the behaviour of the net in such a way that the transition c is only enabled if additionally to the usual enabledness conditions of p/t-nets the place p contains at most two tokens. That means, through weighted inhibitor arcs it is tested if the number of tokens in a place does not exceed the inhibitor weight (as an enabledness condition). In the a-priori semantics the respective testing precedes the actual occurrence of the transition. That means the first LSO (from left) can be interpreted as an execution of the pti-net in the following sense: In the initial marking c and two instances of a are concurrently enabled (accordingly there exist no arcs modelling a causal dependency between the respective nodes), because the double occurrence of a produces (at most) two tokens in p. Therefore the occurrence of c is not prohibited (because the inhibitor arc (p, c) has the weight two). Moreover, after any occurrence of a the transition b is once enabled leading to the two solid "earlier than" arcs between each a and b. The two events labelled by b are concurrent. It is now important that after the double occurrence of a and one occurrence of b the place p contains three tokens. Thereby c is disabled by the inhibitor arc (p, c), i.e. b and c cannot occur in the order $b \to c$ (and therefore b and c are also not concurrent). However, the two transitions can occur synchronously, because in this situation the testing procedure (through the inhibitor arc (p, c)) precedes the occurrence procedure according to the a-priori rule. Thus, it precedes the enhancement of the number of tokens in p from two to three tokens through b. Furthermore, the occurrence in order $c \to b$ is obviously possible. Altogether, this behaviour of the b-labelled events and c can be described as follows: c cannot occur later than b or abbreviated c "not later than" b leading to dashed arcs between c and b in each case. Thus, an execution of a pti-net is an LSO, whose events are labelled with transition names, such that all transitions can occur in the given ordering and concurrency relations.

Technically executions will be defined as enabled LSOs. We propose a definition of enabledness for LSOs generalizing consistently the notion of enabled LPOs. Then every pti-net has assigned a set of executions (enabled LSOs). These describe the complete non-sequential behaviour of the pti-net, i.e. all possible causality and concurrency relationships between transition occurrences. Analogously to the notion of a partial language as a set of (non-isomorphic) LPOs we denote a set of (non-isomorphic) LSOs as a stratified language. Therefore, the non-sequential behaviour of a pti-net represented through the set of all executions of the net is a stratified language. The respective (scenario-based) synthesis problem can be formulated as follows:

Given: A stratified language $\mathcal{L}$ over a finite set of labels.
Searched: A pti-net whose set of executions coincides with the given language $\mathcal{L}$, if such a net exists.

As mentioned, for the less general problem with a partial language as the given behavioural model and a p/t-net as the searched system model the problem was solved in [10] applying the so called theory of regions. A region of a partial language defines a place by determining the initial marking of that place and the weights on each flow arc leading to and coming from a transition. A region of a stratified language additionally has to determine the weights of each inhibitor arc leading to a transition. It turns out that the notion of regions of stratified languages can be based on the notion of regions of partial languages. More precisely, omitting the "not later than" relation of all LSOs of a stratified language yields a set of LPOs forming the partial language underlying the given stratified language. To define regions of stratified languages we start with regions of the underlying partial language ignoring inhibitor arcs and complement these by "possible inhibitor arcs" as they are called in [2]. In this aspect the approach is similar as in [2,11,12] (where the authors started with classical regions of (step) transition systems and complemented these by "possible inhibitor arcs"). Roughly speaking, we add a "possible inhibitor arc" if in each possible intermediate marking state when executing a specified LSO subsequent events are not prohibited by this inhibitor arc. The identification of such inhibitor arcs is more complicated than for elementary nets and (step) transition systems (considered in [2,11,12]). On the one hand we have to regard weighted inhibitor arcs. On the other hand the marking states critical for the inhibitor tests are not directly modelled in LSOs (in contrast to transition systems). Having solved this problem, as the main theorem of this paper we show that the regions of a stratified language exactly define all feasible pti-net places (w.r.t. this stratified language). Thus, the regions of a stratified language define the saturated feasible pti-net. This net has a minimal set of executions including the given stratified language (among all pti-nets) and therefore solves the synthesis problem or is the best approximation if no solution exists. This solves the synthesis problem satisfactory from the theoretical point of view (for the considered setting). Practical algorithmic considerations are a topic of further research (see also the conclusion for a brief discussion).

The paper is structured as follows: First the basic notions of pti-nets and enabled LSOs are introduced (section 2). Then in section 3 the general fundamentals of the region based synthesis are developed and in section 4 the theory of regions is concretely evolved for the formulated synthesis problem.

2 Pti-nets

In this section we recall the basic definitions of *pti-nets* and introduce *enabled stratified order structures* as *executions* of pti-nets (leading to a formal model of scenario-based non-sequential semantics of pti-nets).

By $\mathbb{N}$ we denote the non-negative integers and by $\mathbb{N}^+$ the non-negative integers excluding 0. We additionally denote ω an infinite integer, i.e. $n < \omega$ for $n \in \mathbb{N}$. Given a finite set A, the identity relation on A is denoted by id_A and the set of all multi-sets over A is denoted by $\mathbb{N}^A$ (for $m \in \mathbb{N}^A$ we write $a \in m$ if $m(a) > 0$).

A *net* is a triple (P, T, F), where P is a set of *places*, T is a finite set of *transitions*, satisfying $P \cap T = \emptyset$, and $F \subseteq (P \cup T) \times (T \cup P)$ is a *flow relation*. Let (P, T, F) be a net and $x \in P \cup T$ be an element. The *preset* $\bullet x$ is the set $\{y \in P \cup T \mid (y, x) \in F\}$, and the *post-set* $x\bullet$ is the set $\{y \in P \cup T \mid (x,y) \in F\}$. Given a set $X \subseteq P \cup T$, this notation is extended by $\bullet X = \bigcup_{x\in X} \bullet x$ and $X\bullet = \bigcup_{x\in X} x\bullet$.

A *place/transition net* (shortly *p/t-net*) is a quadruple (P,T,F,W), where (P,T,F) is a net and $W : F \to \mathbb{N}^+$ is a *weight function*. We extend the weight function W to pairs of net elements $(x,y) \in (P \times T) \cup (T \times P)$ with $(x,y) \notin F$ by $W(x,y) = 0$.

Definition 1 (Pti-net). *A* pti-net N *is a five-tuple* (P,T,F,W,I)*, where* (P,T,F,W) *is a p/t-net and* $I : P \times T \to \mathbb{N} \cup \{\omega\}$ *is the* weighted inhibitor relation. *If* $I(p,t) \neq \omega$*, then* $(p,t) \in P \times T$ *is called* (weighted) inhibitor arc *and* p *is an* inhibitor place of t.

A *marking* of a pti-net $N = (P,T,F,W,I)$ is a function $m : P \to \mathbb{N}$ (a multi-set over P) assigning a number of tokens to each place. A transition t can only be executed if (in addition to the well-known p/t-net occurrence rule) each $p \in P$ contains at most $I(p,t)$ tokens. In particular, if $I(p,t) = 0$ then p must be empty. $I(p,t) = \omega$ means that t can never be prevented from occurring by the presence of tokens in p. In diagrams, inhibitor arcs have small circles as arrowheads. Just as normal arcs, inhibitor arcs are annotated with their weights. Now however, the weight 0 is not shown. A *marked pti-net* is a pair (N, m_0), where N is a pti-net and m_0 is a marking of N called *initial marking*. Figure 1 shows a marked pti-net.

According to the a-priori semantics of pti-nets, the inhibitor test for enabledness of a transition precedes the consumption and production of tokens in places. A multi-set (a step) of transitions is (synchronously) enabled in a marking if in this marking each transition in the step obeys the inhibitor constraints before the step is executed.

Definition 2 (Occurrence rule, a-priori semantics). *Let* $N = (P,T,F,W,I)$ *be a* pti-net*. A multi-set of transitions* τ *(a* step*) is* (synchronously) enabled to occur in a marking m (w.r.t. the a-priori semantics) *if* $m(p) \geq \sum_{t\in\tau} \tau(t)W(p,t)$ *and* $m(p) \leq I(p,t)$ *for each transition* $t \in \tau$ *(for every place* $p \in P$*).*

The *occurrence* of a step (of transitions) τ leads to the new marking m' defined by $m'(p) = m(p) - \sum_{t\in\tau} \tau(t)(W(p,t) - W(t,p))$ (for every $p \in P$). We write $m \xrightarrow{\tau} m'$ to denote that τ is enabled to occur in m and that its occurrence leads to m'. A finite sequence of steps $\sigma = \tau_1 \ldots \tau_n,\ n \in \mathbb{N}$ is called a *step occurrence sequence enabled in a marking* m *and leading to* m_n, denoted by $m \xrightarrow{\sigma} m_n$, if there exists a sequence of markings $m_1, \ldots, m_n$ such that $m \xrightarrow{\tau_1} m_1 \xrightarrow{\tau_2} \ldots \xrightarrow{\tau_n} m_n$. A step occurrence sequence can be understood as a possible single *observation* of the behaviour of a pti-net, where the occurrences of transitions in one step are observed *at the same time* or *synchronously*. We use the notions for (marked) pti-nets also for (marked) p/t-nets (a p/t-net can be understood as a pti-net with an inhibitor relation which equals ω).

We now introduce *stratified order structures* (*so-structures*) to model executions of pti-nets as sketched in the introduction. We start with some basic notions preparative to the definition of so-structures. A *directed graph* is a pair $(V, \to)$, where V is a finite *set of nodes* and $\to \subseteq V \times V$ is a binary relation over V called the *set of arcs*. As usual, given a binary relation $\to$, we write $a \to b$ to denote $(a,b) \in \to$. Two nodes $a, b \in V$ are called *independent* w.r.t. the binary relation $\to$ if $a \not\to b$ and $b \not\to a$. We denote the set of all pairs of nodes independent w.r.t. $\to$ by $\mathrm{co}_{\to} \subseteq V \times V$. A *partial order* is a directed graph $\mathrm{po} = (V, <)$, where $<$ is an irreflexive and transitive binary relation on V. If $\mathrm{co}_< = id_V$ then $(V, <)$ is called *total*. Given two partial orders $\mathrm{po}_1 = (V, <_1)$ and $\mathrm{po}_2 = (V, <_2)$, we say that po_2 is a *sequentialization* (or *extension*) of po_1 if $<_1 \subseteq <_2$.

So-structures are, loosely speaking, combinations of two binary relations on a set of nodes (interpreted as *events*), where one is a partial order representing an "earlier than" relation and the other represents a "not later than" relation. Thus so-structures describe finer causalities than partial orders. Formally, so-structures are *relational-structures* (*rel-structures*) satisfying certain properties. A rel-structure is a triple $\mathcal{S} = (V, \prec, \sqsubset)$, where V is a finite set (of *events*), and $\prec \subseteq V \times V$ and $\sqsubset \subseteq V \times V$ are binary relations on V. A rel-structure $\mathcal{S}' = (V, \prec', \sqsubset')$ is said to be an *extension* (or *sequentialization*) of another rel-structure $\mathcal{S} = (V, \prec, \sqsubset)$, written $\mathcal{S} \subseteq \mathcal{S}'$, if $\prec \subseteq \prec'$ and $\sqsubset \subseteq \sqsubset'$.

Definition 3 (Stratified order structure [6]). *A rel-structure $\mathcal{S} = (V, \prec, \sqsubset)$ is called* stratified order structure *(so-structure) if the following conditions are satisfied for all $u, v, w \in V$:*

$(C1)\ u \not\sqsubset u.$ $\quad (C3)\ u \sqsubset v \sqsubset w \wedge u \neq w \Longrightarrow u \sqsubset w.$

$(C2)\ u \prec v \Longrightarrow u \sqsubset v.$ $\quad (C4)\ u \sqsubset v \prec w \vee u \prec v \sqsubset w \Longrightarrow u \prec w.$

In figures $\prec$ is graphically expressed by solid arcs and $\sqsubset$ by dashed arcs. According to (C2) a dashed arc is omitted if there is already a solid arc. Moreover, we omit arcs which can be deduced by (C3) and (C4). It is shown in [6] that $(V, \prec)$ is a partial order. Therefore so-structures are a generalization of partial orders which turned out to be adequate to model the causal relations between events of pti-nets under the a-priori semantics. In this context $\prec$ represents the ordinary "earlier than" relation (as for p/t-nets) while $\sqsubset$ models a "not later than" relation (see Figure 1 for an example).

For our purposes we have to consider *labelled so-structures* (*LSOs*) where the nodes of an so-structure represent transition occurrences of a pti-net (nodes are labelled by transition names as in Figure 1). Formally these are so-structures $\mathcal{S} = (V, \prec, \sqsubset)$ together with a *set of labels* T and a *labelling function* $l : V \to T$. The labelling function l is lifted to a subset Y of V in the following way: $l(Y)$ is the multi-set over T given by $l(Y)(t) = |l^{-1}(t) \cap Y|$ for every $t \in T$. We will use the notations for so-structures also for LSOs as well as for LPOs (since an LPO can be understood as an LSO with $\prec = \sqsubset$). We will consider LSOs only up to isomorphism. Two LSOs $(V, \prec, \sqsubset, l)$ and $(V', \prec', \sqsubset', l')$ are called *isomorphic*, if there is a bijective mapping $\psi : V \to V'$ such that $l(v) = l'(\psi(v))$ for $v \in V$, $v \prec w \Leftrightarrow \psi(v) \prec' \psi(w)$ and $v \sqsubset w \Leftrightarrow \psi(v) \sqsubset' \psi(w)$ for $v, w \in V$. By $[\mathcal{S}]$ we will denote the set of all LSOs isomorphic to $\mathcal{S}$. The LSO $\mathcal{S}$ is said to *represent* the isomorphism class $[\mathcal{S}]$.

As explained, for the modelling of system behaviour the two relations of an LSO are interpreted as "earlier than" resp. "not later than" relation between transition occurrences. If two transition occurrences are in "not later than" relation, that means they can be observed (are allowed to be executed) synchronously or sequentially in one specific order. If two transitions are neither in "earlier than" relation nor in "not later than" relation, they are concurrent and can be observed (are allowed to be executed) synchronously or sequentially in any order. In this sense one LSO "allows" many observations (step sequences). If all these observations are enabled step occurrence sequences, this LSO is called *enabled*. Formally the observations "allowed" by an LSO are defined through so called total linear extensions of the LSO:

Definition 4 (Total linear so-structures). *Let $\mathcal{S} = (V, \prec, \sqsubset)$ be an so-structure, then $\mathcal{S}$ is called* total linear *if* $\text{co}_{\prec} = (\sqsubset \setminus \prec) \cup id_V$*. The set of all* total linear extensions *(or* linearizations*) of an so-structure $\mathcal{S}$ is denoted by $lin(\mathcal{S})$.*

Total linear so-structures are maximally sequentialized in the sense that no further $\prec$- or $\sqsubset$- relations can be added maintaining the requirements of so-structures according to Definition 3 (adding a $\prec$- or $\sqsubset$- relation leads to causal relations of the form $u \sqsubset v \prec u$). Therefore the linearizations $lin(\mathcal{S})$ of an so-structure $\mathcal{S}$ are its maximal extensions.

With this definition the set of step sequences (observations) "allowed" by an LSO is defined as the set of step sequences extending the LSO (that means emerging from adding causality to the LSO). A step sequence can be easily interpreted as a total linear LSO: Each step corresponds to a set of events labelled by transitions (transition occurrences) which are in "not later than" relation with each other representing synchronous transition occurrences. Transition occurrences in different steps are ordered in appropriate "earlier than" relation. Formally, for a sequence of transition steps $\sigma = \tau_1 \dots \tau_n$ define the total linear LSO $\mathcal{S}_\sigma = (V, \prec, \sqsubset, l)$ *underlying* σ by: $V = \bigcup_{i=1}^{n} V_i$ and $l : V \to T$ with $l(V_i)(t) = \tau_i(t)$, $\prec = \bigcup_{i<j} V_i \times V_j$ and $\sqsubset = ((\bigcup_i V_i \times V_i) \cup \prec) \setminus id_V$. ($\mathcal{S}_\sigma$ is total linear because $\text{co}_{\prec} = \bigcup_{i=1}^{n} V_i \times V_i$). Altogether a step sequence σ is "allowed" by an LSO $\mathcal{S}$ if $\mathcal{S}_\sigma \in lin(\mathcal{S})$. For example the step sequences respectively observations "allowed" by the third LSO in Figure 1 can be characterized as follows: To each of the step sequences $cabb$, $(c + a)bb$, $acbb$ and $a(b + c)b$ an a has to be added either to one of the steps or representing a one-element step ordered in any position of the sequence. Any such possibility has to be regarded leading to 29 different "allowed" step sequences, e.g. including $cabab$, $(c + 2a)bb$, $2acbb$ or $a(b + c)(a + b)$.

Note that for each total linear LSO $\mathcal{S} = (V, \prec, \sqsubset, l)$ there is a step sequence σ such that $\mathcal{S}$ and $\mathcal{S}_\sigma$ are isomorphic. That means total linear LSOs can be interpreted as step sequences and the "allowed" observations of an LSO $\mathcal{S}$ in this sense are exactly the step sequences given by $lin(\mathcal{S})$.

Now we define enabled LSOs w.r.t. a marked pti-net as LSOs whose "allowed" observations are also "allowed" in the marked pti-net. More technically this means that any step sequence extending the LSO is enabled in the marked pti-net. Such an enabled LSO is called an execution of the marked pti-net.

Definition 5 (Enabled LSO). *Let (N, m_0), $N = (P, T, F, W, I)$, be a marked pti-net. An LSO $\mathcal{S} = (V, \prec, \sqsubset, l)$ with $l : V \to T$ is called* enabled (to occur) w.r.t. (N, m_0) (in the a-priori semantics) *if the following statement holds: Each finite step sequence $\sigma = \tau_1 \dots \tau_n$ with $\mathcal{S}_\sigma \in lin(\mathcal{S})$ is an enabled step occurrence sequence of (N, m_0).*

In other words an LSO is enabled if and only if it is consistent with the step semantics. This reflects the general idea for the modelling of non-sequential system behaviour that scenarios which are consistent with the non-sequential occurrence rule represent executions.[4] The presented definition is a proper generalization of the notion of enabled

[4] Another possibility for the definition of enabled LSOs is to consider sequences of concurrent steps of synchronous steps instead of sequences of synchronous steps. But both notions are equivalent, as discussed in [7].

LPOs: An LPO $\mathrm{lpo} = (V, \prec, l)$ with $l : V \to T$ is *enabled to occur in a marking* m of a marked p/t-net (P, T, F, W, m_0) if each step sequence which extends (sequentializes) lpo is a step occurrence sequence enabled in m_0. Since in LPOs concurrent and synchronous transition occurrences are not distinguished, here a step is considered as a set of events labelled by transitions (transition occurrences) which are concurrent.

Now it is possible to formally check that the LSOs from Figure 1 are indeed enabled LSOs w.r.t. the shown pti-net. For example in the case of the third LSO one would have to verify that the 29 step sequences "allowed" by this LSO (these are characterized above) are enabled step sequences of the marked pti-net.

Having defined single executions of marked pti-nets the behavioural model in our setting is defined as follows:

Definition 6 (Stratified language). *Let T be a finite set. A subset $\mathcal{L} \subseteq \{[\mathcal{S}] \mid \mathcal{S}$ is an LSO with set of labels $T\}$ is called* stratified language over T *(in the special case of LPOs it is called* partial language*). The* stratified language of executions $L(N, m_0)$ of a marked pti-net (N, m_0) *is defined as the stratified language consisting of all (isomorphism classes of) executions of* (N, m_0).

In the following we only consider stratified languages over sets T such that every $t \in T$ occurs as a label of some node of the stratified language (without explicitly mentioning this). Moreover, since we regard LSOs only up to isomorphism, we assume for the rest of the paper that a stratified language $\mathcal{L}$ over a finite set of labels is given by a set L of LSOs representing $\mathcal{L}$ in the sense that $[\mathcal{S}] \in \mathcal{L} \iff \exists \mathcal{S}' \in L : [\mathcal{S}] = [\mathcal{S}']$. Note that the stratified language of executions of a marked pti-net (N, m_0) is *sequentialization closed*. That means given an execution $\mathcal{S} \in L(N, m_0)$ of (N, m_0), any sequentialization of $\mathcal{S}$ is also an execution of (N, m_0). This is a simple observation using Definition 5, since sequentializations have a smaller set of linearizations. Moreover, as in the LPO-case, the stratified language of executions of (N, m_0) is *prefix closed*, where prefixes of so-structures are defined as subsets of nodes which are downward closed w.r.t. the $\sqsubset$-relation:

Definition 7 (Prefix). *Let $\mathcal{S} = (V, \prec, \sqsubset)$ be an so-structure and let $V' \subseteq V$ be such that $u' \in V'$, $u \sqsubset u' \Longrightarrow u \in V'$. Then $\mathcal{S}' = (V', \prec|_{V' \times V'}, \sqsubset|_{V' \times V'})$ is called* prefix *of $\mathcal{S}$. We say that the prefix $\mathcal{S}'$ is* defined by V'. *If additionally $(u \prec v \Longrightarrow u \in V')$ for some $v \in V \setminus V'$, then $\mathcal{S}'$ is called* prefix of v (w.r.t. $\mathcal{S}$).

3 The Synthesis Problem

The behaviour of a pti-net is described by its stratified language of executions. Therefore, for a stratified language L the question whether it represents the non-sequential behaviour of a marked pti-net can be formulated. The answer to this question together with a concrete characterization of such a net in the positive case are the central issues of this paper. Technically this synthesis problem can be fixed as follows:

Given: A stratified language L over a finite set of labels.
Searched: A marked pti-net (N, m_0) with $L(N, m_0) = L$ if such (N, m_0) exists.

In the following we outline the synthesis principles of the so called theory of regions. The concrete regions-based synthesis approach for the synthesis problem of pti-nets from stratified languages is developed in the next section.

The transition set T of the searched marked pti-net (N, m_0) is obviously given through the finite set of labels of the stratified language L (equally labelled nodes of LSOs in L represent occurrences of the same transition). Considering the pti-net $N = (\emptyset, T, \emptyset, \emptyset, \emptyset)$ with this transition set and an empty set of places, obviously any LSO in L is an execution of $(N, \emptyset)$. This is clear because in N there are no causal dependencies between the transitions. Therefore, every LSO with labels in T is enabled. On the other hand, there are also a lot of executions of $(N, \emptyset)$ not specified in L, i.e. $L(N, \emptyset) \supsetneq L$. Since we are interested in $L(N, m_0) = L$, we have to restrict the behaviour of (N, m_0) by introducing causal dependencies between transition occurrences. Such dependencies between transitions can (only) be realized by adding places to (N, m_0). Any place (with an initial marking) prohibits a certain set of LSOs from being enabled. The central idea is to add all places to (N, m_0) that do not prohibit LSOs specified in L from being enabled. These places are called *feasible places* and lead to the so called *saturated feasible pti-net* (N, m_0). For this net of course $L(N, m_0)$ still includes L, i.e. the specified LSOs in L are enabled w.r.t. (N, m_0) constructed in this way, while it is still not clear if $L(N, m_0) = L$. But now the marked pti-net (N, m_0) has minimal (w.r.t. set inclusion) non-sequential behaviour $L(N, m_0)$ including L, since all places not prohibiting L are regarded. That means that (N, m_0) is the appropriate candidate for the solution of the synthesis problem. If (N, m_0) does not solve the problem there exists no net solving the problem. This is ensured by construction because any other net solving the synthesis problem in this case would contradict the minimality property of (N, m_0) (since it would have a smaller set of executions including L).

The construction of the saturated feasible pti-net involves the introduction of places. Any place consists of an initial marking, a flow and an inhibitor relation to each transition and a flow relation from each transition. Consequently any place p can be defined by the value of its initial marking $m_0(p)$ together with the flow and inhibitor weights $W(p, t), W(t, p)$ and $I(p, t)$ for any transition $t \in T$ as depicted on the left of Figure 2 (a flow weight of 0 respectively an inhibitor weight of ω means that no such arc exists, compare section 2). Any place p restricts the behaviour of a marked pti-net by prohibiting a certain set of LSOs from being enabled. This set of LSOs prohibited by p does only depend on this place p. That means it does not matter if we consider the one-place net having p as its only place or a marked pti-net with a lot of places including p. More precisely, an LSO is enabled w.r.t. a marked pti-net (N, m_0), $N = (P, T, F, W, I)$, if and only if it is enabled w.r.t. every respective one-place net (for every $p \in P$). Regarding a given stratified language L the behavioural restriction of such a place p can be *feasible* or *non-feasible*, i.e. too restrictive, in the following sense (F, W, I and m_0 are determined by the definition of p – an example of a feasible and a non-feasible place is illustrated in Figure 2):

– *Non-feasible places p w.r.t. L*: There exists an LSO $\mathcal{S} \in L$, which is not enabled w.r.t. the one-place pti-net (N, m_0), $N = (\{p\}, T, F, W, I)$, i.e. $L \not\subseteq L(N, m_0)$.
– *Feasible places p w.r.t. L*: Every LSO $\mathcal{S} \in L$ is enabled w.r.t. the one-place pti-net (N, m_0), $N = (\{p\}, T, F, W, I)$, i.e. $L \subseteq L(N, m_0)$.

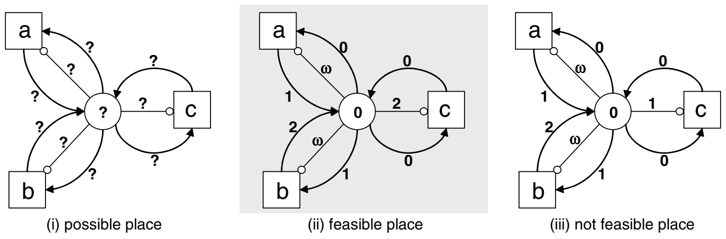


Fig. 2. **(i)** The general structure of a place. **(ii)** A feasible place w.r.t. the stratified language from Figure 1 (it coincides with the place p in Figure 1). **(iii)** A non-feasible place w.r.t. the stratified language from Figure 1. The inhibitor arc to the transition c (in contrast to (ii) with inhibitor weight 1 instead of 2) is causally too restrictive. To verify this recall the considerations in the context of Figure 1 in the Introduction.

Every net solving (positively) the synthesis problem necessarily does not contain a non-feasible place. Therefore the crucial idea is to consider the marked pti-net (N, m_0), $N = (P, T, F, W, I)$, containing exactly all feasible places w.r.t. L. Considering the above explanations this so called *saturated feasible pti-net* (N, m_0) guarantees that any LSO $\mathcal{S} \in L$ is enabled w.r.t. (N, m_0) (called property (A) of the saturated feasible pti-net in the following). Moreover, the saturated feasible pti-net (N, m_0) can have more executions than specified by L, but there is no marked pti-net with a smaller set of executions including L (called property (B) of the saturated feasible pti-net in the following). This is true because any other net (N', m_0') whose set of executions $L(N', m_0')$ includes L mandatory has less places than (N, m_0) since it may only contain feasible places (it holds $L(N', m_0') \supseteq L(N, m_0)$ if (N', m_0') has less places than (N, m_0)).

Definition 8 (Saturated feasible pti-net). *Let L be a stratified language over the set of labels T, then the marked pti-net (N, m_0), $N = (P, T, F, W, I)$, such that P is the set of all places feasible w.r.t. L is called* saturated feasible pti-net (w.r.t. L).

The saturated feasible pti-net (N, m_0) w.r.t. L in general has infinitely many (feasible) places. It fulfills (A) $L \subseteq L(N, m_0)$ and (B) $L(N, m_0) \subseteq L(N', m_0')$ for each marked pti-net (N', m_0'), $N' = (P', T, F', W', I')$, fulfilling $L \subseteq L(N', m_0')$ (thus fulfilling (A)). For the solution of the synthesis problem it is enough to consider only the saturated feasible pti-net, because either this net solves the synthesis problem or there is no solution for the problem:

Theorem 1. *Let L be a stratified language and (N, m_0), $N = (P, T, F, W, I)$, be the saturated feasible pti-net w.r.t. L, then $L(N, m_0) \neq L$ implies $L(N', m_0') \neq L$ for every marked pti-net (N', m_0'), $N' = (P', T, F', W', I')$.*

Property (B) even tells us more than this theorem: In the case $L(N, m_0) \neq L$, $L(N, m_0)$ is the best upper approximation to L. That means the saturated feasible pti-net is the best approximation to a system model with non-sequential behaviour given by L among all marked pti-nets allowing the behaviour specified by L.

Altogether, in order to solve the synthesis problem in our setting, we want to calculate the saturated feasible pti-net. Therefore we are interested in a characterization of feasible places based on L that leads to an effective calculation method for feasible places. In the p/t-net case such a characterization was developed for behavioural models w.r.t. sequential semantics and step semantics [1] with the notion of *regions* of the behavioural model. These approaches were generalized in [10] to partial languages. In the latter case it was shown that every region of a partial language L defines a place such that

(1) Each place defined by a region of L is feasible w.r.t. L.
(2) Each place feasible w.r.t. L can be defined by a region of L.

In [10] we used a slightly different terminology as in this paper. In particular, we did not use the notion of feasible places there but their characterization by the so called *token flow property*. To prove the mentioned results we assumed that the set L of LPOs representing the given partial language satisfies certain technical requirements. More precisely, L was assumed to be prefix and sequentialization closed, since such partial languages are the only candidates as models of the non-sequential behaviour of a marked p/t-net. Moreover, we required that LPOs which are in conflict (describe alternative executions) have disjoint node sets (for the exact formal definitions we refer to [10]). We showed that such representations always exist. Since our approach is based on the results in [10], we require analogous technical properties for the representation L of the specified stratified language. As in the p/t-net case it is no restriction for the synthesis problem to consider only such representations of prefix and sequentialization closed stratified languages.

In examples we will always give such L by a *set of minimal LSOs* of L (minimal LSOs of L are not an extension of some other LSO in L), such that each LSO in L is an extension of some prefix of one of these minimal LSOs. Thus every set of LSOs which are not extensions of each other can be interpreted as a representation of a stratified language by minimal LSOs. For example the four LSOs in Figure 1 represent the stratified language that exactly coincides with the stratified language of executions given by the non-sequential behaviour of the pti-net on the left of Figure 1.

The main aim of this paper is the generalization of the region definition to our setting such that (1) and (2) hold for stratified languages L w.r.t. pti-nets. With such a notion of regions based on stratified languages, the saturated feasible pti-net w.r.t. a stratified language L is directly defined by the set of all regions: Every region of L defines a place of the saturated feasible pti-net. This is the basis for effective solution algorithms for the synthesis problem considered in this paper: In the case of [1] as well as [10] (for the approach of [10] we developed a respective algorithm for finite partial languages in the recent paper [9]) algorithms for the calculation of finite representations of the set of regions were deduced. In the conclusion we argue why this is also possible in our setting. A detailed elaboration of this topic will be the issue of further publications.

4 Regions of Stratified Languages (w.r.t. Pti-nets)

In this section we extend the notion of regions known for partial languages and p/t-nets to the setting of pti-nets. In [10] it is shown that the regions of a partial language in the context of p/t-nets exactly correspond to the feasible places w.r.t. the partial language. Our aim is to show the same for stratified languages and pti-nets.

Fix a marked pti-net (N, m_0), $N = (P, T, F, W, I)$, and an LSO $\mathcal{S} = (V, \prec, \sqsubset, l)$ with $l : V \to T$. Assume that $\mathcal{S}$ is enabled to occur w.r.t. (N, m_0). Since the inhibitor relation I of (N, m_0) restricts the behaviour of the underlying p/t-net (P, T, F, W, m_0), $\mathcal{S}$ is then also enabled w.r.t. the p/t-net $(N', m_0) = (P, T, F, W, m_0)$ underlying N. In a p/t-net, transitions which can be executed synchronously can also be executed concurrently. Therefore, also the LPO $\mathrm{lpo}_{\mathcal{S}} = (V, \prec, l)$ (omitting the "not later than" relation) underlying $\mathcal{S}$ is enabled w.r.t. the p/t-net (N', m_0). Altogether, for a set of enabled LSOs w.r.t. (N, m_0), the LPOs underlying these LSOs are enabled w.r.t. the underlying p/t-net (N', m_0). Considering a one place-net (N, m_0) as in the definition of feasible places, it becomes clear that we have the following necessary condition for a feasible place p w.r.t. a stratified language L: The place p' underlying p defined by omitting the inhibitor relation from p is feasible w.r.t. the underlying partial language consisting of the LPOs underlying the LSOs from L.

Lemma 1. *Let L be a stratified language with transition labels T and let $L' = \{(V, \prec, l) \mid (V, \prec, \sqsubset, l) \in L\}$ be* the partial language underlying *L. Then for any place p feasible w.r.t. L (in the pti-net context) the place p'* underlying *p, defined by $W(p', t) = W(p, t), W(t, p') = W(t, p), I(p, t) = \omega$ for every $t \in T$ and $m_0(p') = m_0(p)$, is feasible w.r.t. L' (in the pti-net as well as the p/t-net context).*

That means, any place p feasible w.r.t. L can be constructed from a place p' which is feasible w.r.t. the underlying partial language L' and has inhibitor weights $I(p', t) = \omega$ (for every transition $t \in T$) by adding appropriate (respectively feasible) inhibitor weights $I(p, t)$. In particular, every place p feasible w.r.t. L fulfilling $I(p, t) = \omega$ for every transition $t \in T$ is feasible w.r.t. L'. On the other hand also the reverse holds: Every place p' feasible w.r.t. L' is feasible w.r.t. L because the enabledness of the underlying LPOs from L' w.r.t. the one place net defined by p' implies the enabledness of the original LSOs from L w.r.t. this net (since they have more causal ordering). Consequently, the sets of feasible places p with $I(p, t) = \omega$ for every $t \in T$ coincide for L and L'. Since L' is a partial language and the restriction $I(p, t) = \omega$ corresponds to p/t-net places, we can characterize these places using the theory of regions for partial languages and p/t-nets from [10]: The p/t-net places feasible w.r.t. the partial language L' are exactly the places defined by regions of L'. Thus, we can characterize the set of all feasible places p w.r.t. L fulfilling $I(p, t) = \omega$ for every $t \in T$ with the regions theory of [10]. Moreover, from Lemma 1 we know that any further place feasible w.r.t. L having inhibitor weights not equal to ω coincides with one of these feasible places p (fulfilling $I(p, t) = \omega$ for every $t \in T$) except of the inhibitor weights.

As a consequence, the regions definition in our setting is based on the regions definition for partial languages and p/t-nets. More precisely, we start with p/t-net regions of the underlying partial language L'. This leads to the set of feasible places p fulfilling $I(p, t) = \omega$ for every $t \in T$ as described above. Then we examine for each such p which other inhibitor weight combinations $I(p, t)$ (preserving the flow relation and the initial marking) also lead to feasible places. For this we use that incrementing an inhibitor weight alleviates the behavioural restriction of the respective inhibitor arc. In particular the set of enabled step sequences and the set of executions increases. Consequently incrementing the inhibitor weight of a feasible place obviously leads again to a feasible place (since the resulting places are causally less restrictive). That means, considering a feasible place p as above with $I(p,t) = \omega$ for every $t \in T$, there is a minimal value $I_{min}(p,t) \in \mathbb{N} \cup \{\omega\}$ for the inhibitor weight to every single transition t such that the following holds: p is still feasible if we change $I(p,t)$ so that $I(p,t) \geq I_{min}(p,t)$ and no more feasible if we change $I(p,t)$ so that $I(p,t) < I_{min}(p,t)$ (preserving $I(p,t') = \omega$ for every $t' \in T \setminus \{t\}$). Now it is important that we can combine these different minimal values $I_{min}(p,t)$ (for different $t \in T$) to one global lower bound in the following sense: Preserving the flow relations and the initial marking, p is feasible if $I(p,t) \geq I_{min}(p,t)$ for every $t \in T$ and p is non-feasible if $I(p,t) < I_{min}(p,t)$ for one $t \in T$. This combination to one global bound is possible because, given a fixed flow relation, the inhibitor arcs have no causal interrelation between each other. That means it is possible to check the enabledness of an LSO by testing the enabledness w.r.t. the inhibitor arcs one by one. Altogether, the set of feasible places w.r.t. a stratified language L can be defined by the set of p/t-net places (places p with $I(p,t) = \omega$ for every $t \in T$) feasible w.r.t. L together with a global lower bound for the inhibitor weights of each such p/t-net place. Since the feasible p/t-net places p can be characterized by the regions definition for partial languages and p/t-nets, we first recall the regions definition of [10]. Based on this regions definition we then identify the lower inhibitor weight bounds $I_{min}(p,t)$ for the respective places p which then leads to the set of all feasible places w.r.t. L. This generalizes the definition of regions from [10].

The idea of defining regions for partial languages in [10] is based on the notion of *token flow functions*: If two events v and v' are ordered in an LPO $\mathrm{lpo} = (V, <, l)$ – that means $v < v'$ – this specifies that the corresponding transitions $l(v)$ and $l(v')$ are causally dependent in the sense of an "earlier than" relation. In a p/t-net such a causal dependency arises exactly if the occurrence of the transition $l(v)$ produces tokens in a place, which are consumed by the occurrence of the other transition $l(v')$. Such a place will be defined by a token flow function x: Assign to every edge (v, v') of lpo a natural number $x(v, v')$ representing *the number of tokens which are produced by the occurrence of* $l(v)$ *and consumed by the occurrence of* $l(v')$ *in the place to be defined*. Thus, a token flow function x describes the flow weights of a respective place. Additionally the initial and final marking of the place have to be regarded. Therefore, we extend an LPO lpo by an *initial and final event*, representing transitions producing the initial marking of the place to be defined and consuming the final marking of the place to be defined (after the occurrence of lpo). This leads to the $\star$-*extension* $\mathrm{lpo}^\star = (V^\star, <^\star, l^\star)$ of lpo defined by $V^\star = (V \cup \{v_0, v_{\max}\})$, $v_0, v_{\max} \notin V$, $\prec^\star = \prec \cup (\{v_0\} \times V) \cup (V \times \{v_{\max}\}) \cup \{(v_0, v_{\max})\}$, $l^\star(v_0), l^\star(v_{\max}) \notin l(V)$, $l^\star(v_0) \neq l^\star(v_{\max})$ and $l^\star|_V = l$ (v_0 is the *initial event of* lpo and $v_{\max}$ the *final event of* lpo). By defining the token flow function on the edges of $\mathrm{lpo}^\star$ (instead of lpo) also the initial and final marking can be specified.

The natural numbers assigned to the arcs of $\mathrm{lpo}^\star$ by x represent the consumed and produced tokens of the involved transitions in the respective place (whereas the tokens produced by the initial event are interpreted as the initial marking and the tokens consumed by the final event as the final marking). Since the consumed and produced tokens of a transition in a fixed place is given by the flow weights W, we can define the flow weights of the place by x. Clearly, a necessary condition for the definition of W is that equally (with the same transition) labelled events should produce and consume the same overall number of tokens w.r.t. x. The number of tokens produced by an event v of an LPO $\mathrm{lpo}^\star = (V^\star, <^\star, l^\star)$ is called the *outtoken flow of* v *(w.r.t.* lpo *and* x*)* defined by $\mathrm{Out}_{\mathrm{lpo}}(v, x) = \sum_{v <^\star v'} x(v, v')$. The outtoken flow $\mathrm{Out}_{\mathrm{lpo}}(v_0, x)$, which by construction represents the initial marking of the place to be defined by x, is called the *initial token flow of* lpo *(w.r.t.* x*)*. The number of tokens consumed by an event v of an LPO $\mathrm{lpo}^\star = (V^\star, <^\star, l^\star)$ is called the *intoken flow of* v *(w.r.t.* lpo *and* x*)* defined by $\mathrm{In}_{\mathrm{lpo}}(v, x) = \sum_{v' <^\star v} x(v', v)$.

For the definition of the token flow function we not only have to regard one LPO, but a partial language L' over T. Thus we have to consider token flow functions on a set of LPOs. The central property that equally labelled events should produce and consume the same number of tokens has to be extended spanning all LPOs of the given partial language in this situation. Furthermore, since the initial marking has to be unique, the number of tokens produced by the initial event has to coincide for all regarded LPOs.

Formally we consider a $\star$-extension $\mathrm{lpo}^\star = (V^\star, <^\star, l^\star)$ of each $\mathrm{lpo} \in L'$ such that (i) for each two LPOs $(V, <, l), (V', <', l) \in L'$ $l^\star(v_0) = (l')^\star(v_0)$ and (ii) $l^\star(v_{\max}) \neq (l')^\star(v_{\max})$ ($\notin T$) for each two distinct $(V, <, l), (V', <', l') \in L'$. Then the set $(L')^\star = \{\mathrm{lpo}^\star \mid \mathrm{lpo} \in L'\}$ is called *$\star$-extension of* L'. We denote $E_{(L')^\star} = \bigcup_{(V^\star, <^\star, l^\star) \in (L')^\star} <^\star$ as the set of edges of all $\star$-extensions of LPOs in L'. A token flow function x of L' is a function assigning natural numbers to every edge in $E_{(L')^\star}$, such that the tokens produced and consumed by equally labelled events coincide.

Definition 9 (Token flow function of a partial language). *Let L' be a partial language, then a function $x : E_{(L')^\star} \to \mathbb{N}$ is called* token flow function *of L', if for all $\mathrm{lpo} = (V, <, l), \mathrm{lpo}' = (V', <', l') \in (L')^\star$ and for all $v \in V^\star, v' \in V'^\star$ there holds: $l(v) = l'(v') \Longrightarrow (\mathrm{In}_{\mathrm{lpo}}(v, x) = \mathrm{In}_{\mathrm{lpo}'}(v', x) \wedge \mathrm{Out}_{\mathrm{lpo}}(v, x) = \mathrm{Out}_{\mathrm{lpo}'}(v', x))$.*

Since we required that the initial events of all LPOs in $(L')^\star$ have the same label, Definition 9 especially implies that the initial token flows of all LPOs in L' are equal. As explained, the coincidence of the intoken and outtoken flow (respectively the consumed and produced tokens) w.r.t. x of equally labelled events allows to define the *corresponding place* p_x *to* x (in the net with transitions given by the node labels T of L') by $W(l(v), p_x) = \mathrm{Out}_{\mathrm{lpo}}(v, x)$, $W(p_x, l(v)) = \mathrm{In}_{\mathrm{lpo}}(v, x)$ and $m_0(p_x) = \mathrm{Out}_{\mathrm{lpo}}(v_0, x)$ for every $\mathrm{lpo} \in L'$ and every node v of lpo. That means the flow weights of p_x are given by the intoken and outtoken flow of the LPO-events and the initial marking by the initial token flow of the LPOs. In [10] the regions of a partial language L' are exactly the token flow functions of L' as defined here. The respective feasible places are the corresponding places.

We are now interested in token flow functions of the partial language L' underlying the given stratified language L. Thereto we formally define a *token flow function of a stratified language* as a token flow function of its underlying partial language:

Definition 10 (Token flow function of stratified languages). *Let L be a stratified language. Then a* token flow function *of L is a token flow function of the partial language $L' = \{(V, \prec, l) \mid (V, \prec, \sqsubset, l) \in L\}$ underlying L.*

In illustrations we annotate each $\prec$-arc of an LSO in L with the value assigned to the respective arc in L' by a token flow function x (the value 0 is not shown). The non-zero values of x assigned to edges starting from v_0 respectively ending in v_{max} are depicted with small arrows without an initial node respectively without a final node. We only consider minimal LSOs of L because the values of a token flow function on the edges of an LSO already constitute the values on edges of prefixes and extensions (as in the LPO-case). Figure 3 sketches an example token flow function of the stratified language from Figure 1 and the respective corresponding place p (with $I(p,t) = \omega$ for all $t \in T$). The intoken and outtoken flow of equally labelled nodes coincide (e.g. all b-labelled nodes have intoken flow 1 and outtoken flow 2 and the initial token flow of all underlying LPOs is 0).

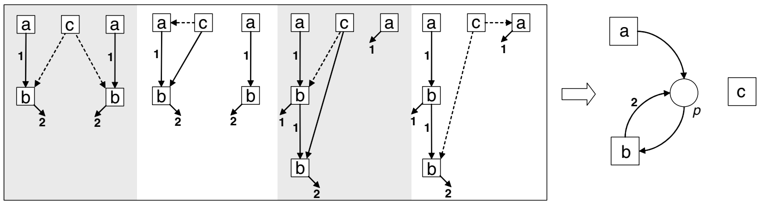


Fig. 3. A token flow function of the stratified language from Figure 1 and the corresponding (feasible) place (with inhibitor weights ω)

According to the above explanations, the places p corresponding to token flow functions x of a stratified language L now exactly define all feasible places w.r.t. L with inhibitor weights ω. In particular, the place p in Figure 3 is feasible w.r.t. the given stratified language. Now it remains to identify the lower bounds $I_{min}(p,t)$ ($t \in T$) for each of these feasible places p (such that $I(p,t) \geq I_{min}(p,t)$ for every $t \in T$ still leads to a feasible place p but $I(p,t) < I_{min}(p,t)$ for some $t \in T$ leads to a non-feasible place p). These minimal possible inhibitor weights $I_{min}(p,t)$ have to be detected with the token flow function x of L. The strategy is as follows: Considering a node v of an LSO $\mathcal{S} = (V, \prec, \sqsubset, l) \in L$ we calculate the minimal inhibitor weight $\mathrm{Inh}(x,v)$ from p to $l(v)$ (where p corresponds to x), such that the occurrence of the transition $l(v)$ according to the causal dependencies given for v in $\mathcal{S}$ is possible. That means, the event v in the context of the scenario given by $\mathcal{S}$ must not be prohibited by an inhibitor arc from p to $l(v)$ in the net if $I(p,l(v)) \geq \mathrm{Inh}(x,v)$, but it is prohibited by such an arc if $I(p,l(v)) < \mathrm{Inh}(x,v)$. Choosing the inhibitor weight $I(p,l(v))$ too small leads to an intermediate marking state of the scenario $\mathcal{S}$ in which a too large number of tokens in p prohibits the occurrence of v. Consequently, in order to determine the minimal inhibitor weight $\mathrm{Inh}(x,v)$ not prohibiting v – called *inhibitor value* of v (w.r.t. x) in the following – it is necessary to calculate the numbers of tokens in p for all intermediate states in which v can occur according to $\mathcal{S}$. Such states are exactly defined by prefixes of v. The maximum of all these possible numbers of tokens in p in such a prefix-state then defines the inhibitor value $\mathrm{Inh}(x,v)$ of v, because according to the scenario $\mathcal{S}$ the transition $l(v)$ should be enabled in each of these token allocations of p. The number of tokens in p in one such prefix-state can be calculated by the token flow function x. The respective number of tokens is given by the number of tokens in p after the execution of the prefix in the corresponding one-place net, called the *final marking of the prefix w.r.t.* x. By construction, the values of x on $\prec^\star$-edges between events of the prefix correspond to tokens which are produced and consumed in p by events in this prefix. On the other hand, the values of x on $\prec^\star$-edges from events of the prefix to events subsequent to the prefix correspond to tokens which are produced by events in the prefix and remain in p after the execution of the prefix. Consequently, the final marking of a prefix can be determined by adding the values of x on $\prec^\star$-edges leaving the prefix.

Definition 11 (Final marking of prefixes). *Let L be a stratified language and x be a token flow function of L. Let $\mathcal{S}' = (V', \prec', \sqsubset', l')$ be a prefix of $\mathcal{S} = (V, \prec, \sqsubset, l) \in L$ and v_0 be the initial event of* $\mathrm{lpo}^\star_{\mathcal{S}} = (V^\star, \prec^\star, l^\star)$*. The* final marking of $\mathcal{S}'$ (w.r.t. x) *is denoted and defined by* $m_{\mathcal{S}'}(x) = \sum_{u \in V',\, v \notin V',\, u \prec v} x(u,v) + \sum_{v \notin V'} x(v_0, v)$.

The final marking of a prefix w.r.t. x can equivalently be calculated by firing the transitions corresponding to the prefix in the one-place net with the place p defined by x (i.e. it is independent from the concrete token flow distribution x and only dependent on p): $m_{\mathcal{S}'}(x) = \sum_{u \in V',\, v \notin V',\, u \prec v} x(u,v) + \sum_{v \notin V'} x(v_0, v) = \sum_{v \in V' \cup \{v_0\}} (\sum_{v \prec^\star w} x(v,w) - \sum_{w \prec^\star v} x(w,v)) = \mathrm{Out}(v_0, x) - \sum_{v \in V'} (\mathrm{In}(v,x) - \mathrm{Out}(v,x)) = m_0(p) - \sum_{v \in V'} (W(p, l(v)) - W(l(v), p))$ (the first equation follows since the values on edges within V' cancel each other out).

Summarizing, the calculation of $\mathrm{Inh}(x,v)$ is achieved by identifying all prefixes of v and calculating the final marking w.r.t. x for each such prefix. The maximum over all these numbers gives $\mathrm{Inh}(x,v)$; the inhibitor value $\mathrm{Inh}(x,v)$ specifies how small the inhibitor weight $I(p, l(v))$ may minimally be without prohibiting the event v.

Definition 12 (Inhibitor value). *Let L be a stratified language, x be a token flow function of L and v be an event of an LSO $\mathcal{S} \in L$. The* inhibitor value $\mathrm{Inh}(x,v)$ *of v w.r.t. x is defined by $Inh(x,v) = \max\{m_{\mathcal{S}'}(x) \mid \mathcal{S}'$ is prefix of v w.r.t. $\mathcal{S}\}$.*

Figure 4 shows the token flow function from Figure 3 supplemented with the inhibitor values of all nodes (depicted in circles attached to the nodes). For example, consider the c-labelled node of the first LSO (from left). This node has four prefixes: the empty prefix with final marking 0, two prefixes consisting of one a-labelled node each with final marking 1 and a prefix with both a-labelled nodes and final marking 2.

Having determined $\mathrm{Inh}(x,v)$ for all nodes v of all LSOs in L one can specify the minimal inhibitor weight $I(p,t)$ from p to some transition t such that no t-labelled event

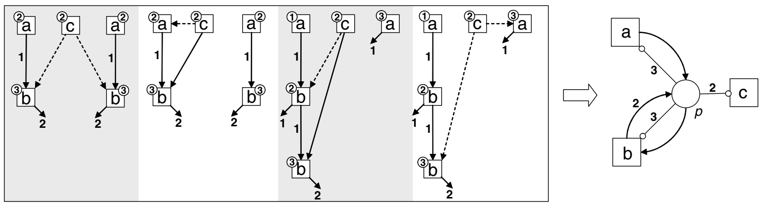


Fig. 4. The token flow function from Figure 3 supplemented with the inhibitor values of all LSO nodes and the feasible place corresponding to the respective region with minimal inhibitor weights

is prohibited by the supremum of all $\text{Inh}(x, v)$ for events v labelled by t. This leads to $I_{min}(p, t)$ because the fact that no such t-labelled event is prohibited by the inhibitor weight $I(p, t)$ exactly describes that the place p is still feasible with this inhibitor weight $I(p, t)$ (instead of ω): $I_{min}(p, t) = sup(\{\text{Inh}(x, v) \mid v \in V_L, l(v) = t\} \cup \{0\})$, where $V_L = \bigcup_{(V,\prec,\sqsubset,l)\in L} V$ is the set of all nodes of L. That means we calculate the inhibitor values of all nodes (over all LSOs of L) w.r.t. a given token flow function x using the method described above. The suprema of all inhibitor values of equally labelled nodes lead to the minimal inhibitor weights defining a feasible place w.r.t. L which corresponds to x. These minimal inhibitor weights $I(p, t) = I_{min}(p, t)$ represent the strongest behavioural restriction through inhibitor arcs for the place p defined by x guaranteeing the feasible-property. Thus *regions of stratified languages* w.r.t. pti-nets are defined by token flow functions x (defining p/t-net places) attached with inhibitor weight mappings $\mathbf{I} : T \to \mathbb{N} \cup \{\omega\}$ determining an inhibitor weight to every transition $t \in T$ which exceeds $I_{min}(p, t)$:

Definition 13 (Region). *A* region *of a stratified language L with labels T w.r.t. pti-nets is a tuple $r = (x, \mathbf{I})$ where x is a token flow function of L and $\mathbf{I} : T \to \mathbb{N}\cup\{\omega\}$ is a mapping assigning inhibitor weights to all transitions satisfying $\mathbf{I}(t) \geq sup(\{\text{Inh}(x, v) \mid v \in V_L, l(v) = t\} \cup \{0\})$.*

The place p_r (in a net with transition set T) corresponding to a region $r = (x, \mathbf{I})$ of L *is defined by the flow weights and the initial marking of the place p_x corresponding to the token flow function x (i.e. $W(l(v), p_r) = \text{Out}_{\text{lpo}}(v, x)$, $W(p_r, l(v)) = \text{In}_{\text{lpo}}(v, x)$ and $m_0(p_r) = \text{Out}_{\text{lpo}}(v_0, x)$ for LPOs* lpo *underlying LSOs in L) and the inhibitor weights $I(p_r, t) = \mathbf{I}(t)$ for $t \in T$.*

The token flow function x in Figure 4 together with the mapping $\mathbf{I}$ given by $\mathbf{I}(a) = 3, \mathbf{I}(b) = 3, \mathbf{I}(c) = 2$ defines a region $r = (x, \mathbf{I})$. In fact this is the respective region with minimal inhibitor weights, i.e. $r' = (x, \mathbf{I}')$ is also a region if $\mathbf{I}' \geq \mathbf{I}$ but no region if $\mathbf{I}' \not\geq \mathbf{I}$. On the right the feasible place p corresponding to r is depicted.

The main theorem of this paper showing the consistency of the above regions definition now states (1) and (2) (compare Section 3) in this setting. Its proof essentially uses the definition of the enabledness of an LSO via the enabledness of its linearizations. According to the following lemma the enabledness of an event after some prefix of an LSO can be examined on the set of its linearizations.

Lemma 2. *Let $\mathcal{S} = (V, \prec, \sqsubset)$ be an so-structure, $V' \subseteq V$ and $v \in V$. Then V' defines a prefix of v w.r.t. $\mathcal{S}$ if and only if there is a linearization $\mathcal{S}' \in lin(\mathcal{S})$ such that V' defines a prefix of v w.r.t. $\mathcal{S}'$.*

Proof. The **if**-statement clearly follows from $\mathcal{S}' \supseteq \mathcal{S}$.

For the **only if**-statement we construct a sequence of event-sets $V_1 \ldots V_n$ with $V = V_1 \cup \ldots \cup V_n$ defining $\mathcal{S}'$ through $\prec_{\mathcal{S}'} = \bigcup_{i<j} V_i \times V_j$ and $\sqsubset_{\mathcal{S}'} = ((\bigcup_i V_i \times V_i) \cup \prec_{\mathcal{S}'}) \setminus id_V$ as follows: $V_1 = \{v \in V' \mid \forall v' \in V' : v' \not\prec v\}$, $V_2 = \{v \in V' \setminus V_1 \mid \forall v' \in V' \setminus V_1 : v' \not\prec v\}$ and so on, i.e. we define $V_i \subseteq V'$ as the set of nodes $\{v \in V' \setminus (\bigcup_{j=1}^{i-1} V_j) \mid \forall v' \in V' \setminus (\bigcup_{j=1}^{i-1} V_j) : v' \not\prec v\}$ which are minimal w.r.t. the restriction of $\prec$ onto the node set $V' \setminus (\bigcup_{j=1}^{i-1} V_j)$, as long as $V' \setminus (\bigcup_{j=1}^{i-1} V_j) \neq \emptyset$. Then continue with the same procedure on $V \setminus V' = V \setminus (\bigcup_{j=1}^{i} V_j)$, i.e. $V_{i+1} = \{v \in$

$V \setminus (\bigcup_{j=1}^{i} V_j) \mid \forall v' \in V \setminus (\bigcup_{j=1}^{i} V_j) : v' \not\prec v\}$ and so on. By construction V' is a prefix (of v) w.r.t. $\mathcal{S}'$. A straightforward computation also yields $\mathcal{S}' \in lin(\mathcal{S})$.

Theorem 2. *Given a stratified language L with set of labels T:* **(1)** *Every place corresponding to a region of L is feasible w.r.t. L and* **(2)** *every feasible place w.r.t. L is corresponding to a region of L.*

Proof. **(1):** Let p be corresponding to a region $r = (x, \mathbf{I})$ of L. We have to show that $\mathcal{S} \in L$ is enabled w.r.t. the one-place net (N, m_0) having p as its only place. Since x is a token flow function (called region in [10]) of the partial language L' underlying L the main result of [10] tells us that the LPO $\text{lpo}_\mathcal{S} \in L'$ underlying $\mathcal{S}$ is enabled w.r.t. the place p_x corresponding to x. Consequently also $\mathcal{S}$ (since $lin(\mathcal{S}) \subseteq lin(\text{lpo}_\mathcal{S})$)) is enabled w.r.t. p_x. In order to show that $\mathcal{S}$ is enabled w.r.t. p (differing from p_x only in the inhibitor weights), we consider a sequence of transition steps $\sigma = \tau_1 \ldots \tau_n$, whose underlying LSO $\mathcal{S}_\sigma$ is a linearization of $\mathcal{S}$. We have to show that σ is an enabled step occurrence sequence of (N, m_0). For this, we show inductively that if $\sigma_k = \tau_1 \ldots \tau_k$ is an enabled step occurrence sequence, then τ_{k+1} is a transition step enabled in the marking m reached after the execution of σ_k for $0 \leqslant k \leqslant n-1$. The above considerations ($\mathcal{S}$ enabled w.r.t. p_x) already imply the first condition of Definition 2 that $m(p) \geq \sum_{t \in \tau_{k+1}} \tau_{k+1}(t) W(p,t)$. It remains to verify the condition of Definition 2 that $m(p) \leq I(p,t)$ for each transition $t \in \tau_{k+1}$. If $\mathcal{S}_{\sigma_k} = (V_k, \prec_k, \sqsubset_k, l_k)$ is the LSO underlying σ_k and $\mathcal{S}_\sigma \supseteq \mathcal{S}$ is the LSO underlying σ, then $\mathcal{S}_{\sigma_k}$ is a prefix of an event $v \in V$ with $l(v) = t$ w.r.t. $\mathcal{S}_\sigma$. By Lemma 2, V_k also defines a prefix $\mathcal{S}_k$ of v w.r.t. $\mathcal{S}$. It is enough to show that $m(p) = m_{\mathcal{S}_k}(x)$, since $m_{\mathcal{S}_k}(x) \leq \text{Inh}(x,v) \leq \mathbf{I}(l(v)) = I(p,t)$ (Definitions 12 and 13): $m(p) = m_0(p) - \sum_{i=1}^{k} \sum_{t \in \tau_i} \tau(t)(W(p,t) - W(t,p)) = m_0(p) - \sum_{v \in V_k}(W(p,l(v)) - W(l(v),p)) = m_{\mathcal{S}_k}(x)$ (compare the remarks to Definition 11).

(2): Let p be feasible w.r.t. L. Then, by Lemma 1 the place p' underlying p is feasible w.r.t. the partial language L' underlying L. The main result of [10] now states that there is a token flow function x of L' (called region in [10]) generating p'. We show now that $r = (x, I(p,\cdot))$ is a region of L (according to Definition 13). The first part that x is a token flow function of L is clear since x is a token flow function of L'. It remains to show $I(p,t) \geq sup(\{\text{Inh}(x,v) \mid v \in V_L, l(v) = t\} \cup \{0\})$. For this let $v \in V$ for $\mathcal{S} = (V, \prec, \sqsubset, l) \in L$ with $l(v) = t$ and $\mathcal{S}'$ be a prefix of v defined by V'. We have to show that $m_{\mathcal{S}'}(x) \leq I(p,t)$ (compare Definition 12). By Lemma 2 there is a linearization $\mathcal{S}_{lin}$ of $\mathcal{S}$ such that V' also defines a prefix $\mathcal{S}'_{lin}$ of v w.r.t. $\mathcal{S}_{lin}$. Since $\mathcal{S}$ is enabled w.r.t. the one-place net (N, m_0) having p as its only place, there is an enabled step occurrence sequence $\sigma = \tau_1 \ldots \tau_n$ of (N, m_0) whose underlying LSO $\mathcal{S}_\sigma$ equals $\mathcal{S}_{lin}$. Since prefixes are downward $\sqsubset$-closed, a prefix $\sigma' = \tau_1 \ldots \tau_m$ $(m < n)$ of σ with $l(v) = t \in \tau_{m+1}$ must exist which corresponds to $\mathcal{S}'_{lin}$. In other words, the LSO $\mathcal{S}_{\sigma'}$ underlying σ' equals $\mathcal{S}'_{lin}$. It is enough to show now that $m(p) = m_{\mathcal{S}'}(x)$ for the marking m reached after the execution of σ' in (N, m_0), since $m(p) \leq I(p,t)$ for each transition $t \in \tau_{m+1}$. The necessary computation is as in (1).

Thus the set of all feasible places and therefore a solution for the synthesis problem can be derived from the set of regions.

5 Conclusion

In this paper we introduced the notion of regions for a (possibly infinite) set of LSOs – called stratified language – describing the behaviour of a pti-net. Given a stratified language L, using such regions allows to define the saturated feasible pti-net (N, m_0) w.r.t. L. The set of executions $L(N, m_0)$ of (N, m_0) includes L and is as small as possible with this property.[5] Thus, the contribution of this paper is to solve the synthesis problem satisfactory from the theoretical point of view (for the considered setting). Practical algorithmic considerations are a topic of further research (see also below).

The presented approach carries over to the a-posteriori semantics of pti-nets, whose non-sequential scenario-based behaviour is given by LPOs, i.e. by partial languages. To define regions for partial languages w.r.t. pti-nets, one can analogously start with regions of the partial language from [10] not specifying inhibitor arcs and then assign inhibitor values to each node. Now, these inhibitor values are determined as maxima over all final markings of classical prefixes of nodes of an LPO, where one has to use a slightly different definition of final markings. It is moreover possible to adapt the presented definition of regions to other less general inhibitor net classes, such as p/t-nets with unweighted inhibitor arcs and elementary nets with inhibitor arcs. Thereby in the case of elementary nets one additionally has to regard that a place defined by a region must not carry more than one token in each intermediate state of an LSO. This can be ensured by only allowing final markings of prefixes $\leqslant 1$ (that means by an analogous mechanism as used for the definition of inhibitor arcs). For step transition systems and stratified languages which produce the same language of step sequences, it would be interesting to compare our (adapted) definition of regions for elementary nets with inhibitor arcs and the definition of regions from [11,12]. The relation is not obvious since several different step transition systems may define the same language of step sequences. In general the ideas presented in this paper should also be useful for the consideration of the synthesis problem of other so-structure based net classes (such as nets with read arcs, priorities, reset arcs, etc.) as well as net classes conceptually similar to inhibitor nets (e.g. elementary nets and nets with capacities).

One of course is interested in practical algorithmic solutions of the synthesis problem. Basically the regions approach has the problem that there is an infinite number of feasible places respectively regions of a stratified language. Our recent publication [9] tackles this problem for finite partial languages and p/t-nets, i.e. a special case of the setting in [10]. Thereto the definition of token flow function is translated into a finite integer system of homogenous inequations $\mathbf{A} \cdot \mathbf{x} \geq 0$: The finite vector $\mathbf{x}$ represents the token flow function and the inequations reflect the conditions of Definition 9 and ensure positive token flows ($\mathbf{x} \geq 0$). It is shown that one can calculate a finite set of basis solutions of this system which defines a set of places spanning all feasible places.[6] That means the net consisting only of these finite, algorithmically determinable set of places

[5] Note that such a region based approach is not appropriate to find a pti-net (N, m_0) such that $L(N, m_0) \subseteq L$ and $L(N, m_0)$ is as large as possible.

[6] An alternative approach is to compute finite many regions which "separate" specified behaviour from not specified behaviour. It is possible to deduce appropriate separation properties from the mentioned algorithm. Such an approach leads to a different finite representation of the saturated feasible net.

has the same set of executions as the saturated feasible net. Furthermore an algorithm testing if this net has the behaviour specified by the finite partial language is shown. In the setting of this paper a similar approach for the effective synthesis of pti-nets from finite stratified languages is possible, i.e. it is possible to calculate finitely many basis regions spanning the set of all regions (using an adequate inequation system). The formal evolution and proofs for this approach including complexity issues are one of our recent research projects in this topic.

But this approach still leaves the problem that it does not work for infinite stratified languages. For algorithmic purposes an infinite stratified language first has to be finitely represented. This problem is strongly connected to the similar problem in the case of p/t-nets and partial languages which is one of our central current research fields.

References

1. Badouel, E., Darondeau, P.: On the synthesis of general petri nets. Technical Report 3025, Inria (1996)
2. Busi, N., Pinna, G.M.: Synthesis of nets with inhibitor arcs. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 151–165. Springer, Heidelberg (1997)
3. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Hardware and petri nets: Application to asynchronous circuit design. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 1–15. Springer, Heidelberg (2000)
4. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. part i: Basic notions and the representation problem. Acta Inf. 27(4), 315–342 (1989)
5. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. part ii: State spaces of concurrent systems. Acta Inf. 27(4), 343–368 (1989)
6. Janicki, R., Koutny, M.: Semantics of inhibitor nets. Inf. Comput. 123(1), 1–16 (1995)
7. Juhás, G., Lorenz, R., Mauser, S.: Complete process semantics for inhibitor nets. In: Proceedings of ICATPN 2007 (2007)
8. Kleijn, H.C.M., Koutny, M.: Process semantics of general inhibitor nets. Inf. Comput. 190(1), 18–69 (2004)
9. Lorenz, R., Bergenthum, R., Mauser, S., Desel, J.: Synthesis of petri nets from finite partial languages. In: Proceedings of ACSD 2007 (2007)
10. Lorenz, R., Juhás, G.: Towards synthesis of petri nets from scenarios. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 302–321. Springer, Heidelberg (2006)
11. Pietkiewicz-Koutny, M.: The synthesis problem for elementary net systems with inhibitor arcs. Fundam. Inform. 40(2-3), 251–283 (1999)
12. Pietkiewicz-Koutny, M.: Synthesising elementary net systems with inhibitor arcs from step transition systems. Fundam. Inform. 50(2), 175–203 (2002)
13. 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)
14. 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)
15. Zhou, M., Cesare, F.D.: Petri Net Synthesis for Discrete Event Control of Manufacturing Systems. Kluwer, Dordrecht (1993)

Utilizing Fuzzy Petri Net for Choreography Based Semantic Web Services Discovery*

Peng Men, Zhenhua Duan, and Bin Yu

Institute of Computing Theory and Technology
Xidian University
Xi'an, 710071, P.R. China
pengmen@gmail.com, zhhduan@mail.xidian.edu.cn,
yubin@mail.xidian.edu.cn

Abstract. Semantic Web Services have received great popularity these days for their capability of providing more automatic web services discoveries and compositions. However, the issue of behavior compatibility is still not properly solved. In this paper, we propose a choreography-based formal approach to discover semantic web services that are compatible with their behaviors. The fuzzy petri net is used to perform the modeling, verification, analysis and evaluation for the matchmaking of semantic web services. The requested and provided services are modeled by two fuzzy petri nets combined through a set of combination rules. By analyzing the composite fuzzy petri net, the behavior compatibility can be verified, and the matching degree between requested and provided services can be obtained. With our approach, the success rate and accuracy of the semantic web services discoveries can be improved, and the final services set can be optimized.

Keywords: fuzzy petri net, semantic web services, ontology, matchmaking.

1 Introduction

Web services are the corner stone of the Service Oriented Architecture (SOA), and made it possible to build loose-coupled and platform-independent systems in an Internet scale. A web service performs an encapsulated function and can be described by WSDL [1]. However, the SOA is not able to be realized unless web services can flexibly be composed. To achieve such flexible compositions, the Business Execution Languages for Web Services (BPEL4WS) [2] was co-authored by current major IT players, such as Microsoft and IBM. OWL-S [3] made further attempts by incorporating semantic web technologies to achieve more automatic web services discoveries and compositions.

* This research is supported by the NSFC Grant No. 60373103 and 60433010.

In order to compose web services effectively by selecting the most suitable services, we need to discover web services that can be safely integrated. It is necessary to consider the following issues:

1. The behavior compatibility needs to be solved for OWL-S based discoveries and compositions. A composite service usually offers multiple operations, which must be choreographed according to its conversation protocol, i.e. two semantic web services must be compatible in their behaviors. The behaviors of two web services are compatible if their operations can be successfully invoked in the desired orders specified in related business protocols.
2. Rational evaluating the matching degree between two web services is required. It need evaluate what extent the provided services can meet the requested services. According to the results of evaluating, we can select the most suitable services. The composite web services based on choreography accomplish a series of interactive processes. Thus, for calculating the matching degree between services, not only are the service functions and qualities compared, but also their interactions are considered.

For solving both of the problems, a formal method is proposed in this paper. With this approach, fuzzy petri net (FPN) is utilized modeling the behavior descriptions in requested and provided services. By combining the two FPNs and analyzing the composite FPN, the compatibility of behavior can be validated effectively. Further, using FPN fuzzy reasoning can calculate the matching degree between services. This method not only evaluates the possibility of one atomic operation substituting for another, but also considers the possibility of match between two services in all kinds of interactions. Therefore, the matching degree between services can be comprehensively evaluated.

The rest of the paper is organized as follows. Section 2 describes existing problems in current web services discoveries with a motivating example; Section 3 briefly introduces the FPN and the ontology concept similarity; Section 4 proposes the transformation from the OWL-S process model into an FPN; after that, in Section 5, the matchmaking algorithm is introduced to compose the FPNs describing the two web services into one FPN; In Section 6, the resulted FPN is analyzed to determine if the behaviors of the two web services in question are compatible, and the matching degree between two services is also evaluated; the conclusion and future works are drawn in Section 7.

2 Motivation Example

In this section, a car sale example is used to introduce the notion of our approach. The entire car sale business process is based on the choreography between buyer and seller services. The internal business processes of the two services are described with OWL-S. As depicted in Fig.1, the car sale is performed in the following order:

– The buyer service invokes the operation *SearchCarInfo* of the seller service to inquire whether the desired car is available.

– If the desired car is not available, the operation *NotifyOutofStock* of the buyer service is invoked. Then the buyer service replies the client with a message, and the transaction terminates.
– If the desired car is available, the seller service informs the buyer service by invoking the *NotifyinStock* operation. Then, the buyer service in turn invokes the operations *SetAccount* and *SetPIN* of seller service to pass the *AccountInfo* and *PINInfo* messages, respectively.
– Finally, the seller service passes the order confirmation to the buyer service by invoking the *SetOrderConfirm* operation, and the confirmation message is returned to the client.

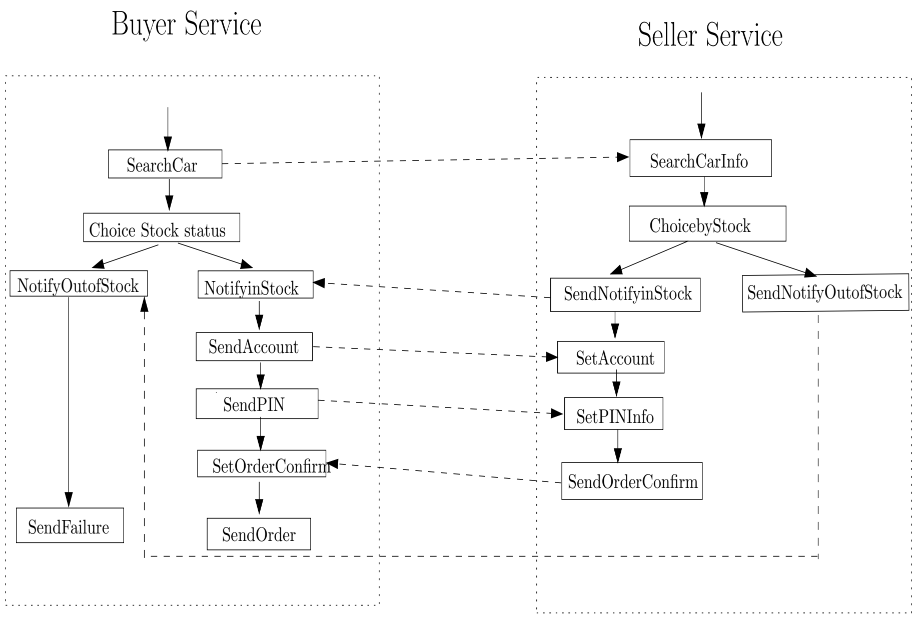


Fig. 1. The interaction between the seller service and the buyer service

The matchmaking of current semantic web services discoveries [4,9,10] widely employs the following strategies: the input of the provided service subsumes the requested input; the output of the provided service subsumes the requested output. However, this kind of pure subsumption based matchmaking has some deficiencies. For instance, the buyer and seller services are not likely to be aware of the existence of the other services in advance, and might be implemented based on different business protocols. This can cause direct conflicts in execution conditions. For example, the seller service requires account information before making any inquiries, while the buyer service prefers to perform free inquiries before any account information is sent. In this case, the business protocol differences cause a deadlock, even the semantics of the two service interfaces are matched. The compatibility of two web services is verified by Alex [5] and Wombacher [6] with petri net and automata, respectively. However, these approaches are not based on semantics. They are performed under the assumption that elements of two message sequences to be compared must come from the same WSDL document to guarantee the same message name represents the same semantics. In [7] Lei transforms OWL-S Process Models into Extended Deterministic Finite Automata (EDFA), and multiplies the resulted EDFAs to determine whether the service described by the OWL-S Process Model is matched. However, the approach can only handle the cases that the semantics of the input and output parameters are exactly matched, without the matching degree being specified.

For the calculation of the matching degree between services, in general, some approaches for comparing the semantics of IOPE or/and QoS are adopted [8]. Nevertheless, this method has not considered the effect of interactions on the matching degree. In fact, for the different business processes, possibly there exist lots of different interactions between services. So, it is necessary to consider all the possible interactions when we calculate the matching degree between two services.

Currently, the matchmaking performed during semantic web services discovery is based on the subsumption relationship among different ontology concepts. However, it might not be satisfied with required demand. For example, when someone wants to find a car seller service, as a result of the match, a truck seller might be found instead of a car seller, and it may turn out to be acceptable. So, We need consider relationship between two arbitrary concepts. In this way, the matchmaking approach can be enhanced.

To solve the above problems, we utilize FPNs to perform web services discovery at the OWL-S process level. First, the OWL-S process is modeled by an FPN; then, we calculate the concept similarities with an ontology matchmaking technique; to do so, the relationship between arbitrary ontology concepts is considered; the two FPNs describing the OWL-S processes are combined into a composite FPN, and the FPN is analyzed to see whether the behaviors of the two processes are compatible; as a result, if they are compatible in behaviors, the matching degree between them is calculated with FPN reasoning rules and the most suitable services can be selected.

3 Preliminaries

3.1 Introduction to FPN

The FPN is firstly introduced in [12] by Looney. It is frequently applied in fuzzy knowledge representation and reasoning about the fuzzy information that cannot be handled by simple petri nets, such as the work introduced in [13,17].

According to the definition in [13], an FPN can be defined as an 8-tuples:

$FPN = (P, T, D, I, O, f, \alpha, \beta)$, where,
$P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places;
$T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions;
$D = \{d_1, d_2, \ldots, d_n\}$ is a finite set of propositions;
$|P| = |D|$

I: $T \rightarrow P^{\infty}$ is an input function, mapping transitions to sets of their input places;

O: $T \rightarrow P^{\infty}$ is an output function, mapping transitions to sets of their output places;

f: $T \rightarrow [0, 1]$ is a function, mapping transitions to real values between zero and one to express the Confidence Factors (CF) of the corresponding rules;

α: $P \rightarrow [0, 1]$ is a function, mapping places to real values between zero and one to denote the degrees of truth of the corresponding propositions;

β: $P \rightarrow D$ is an association function, mapping places to propositions.

When an FPN is used for fuzzy reasoning, a proposition corresponds to a place; a transition is used to represent the relation between causes and effects; a token value stands for the degree of truth of a proposition; each transition has a related CF value representing the strength of the belief in the rule.

In a fuzzy Petri net, the initial token value in a place p_i, $p_i \in P$, is denoted by $\alpha(p_i)$, where $\alpha(p_i) \in$ [0,1]. If $\beta(p_i)$=d_i, $\alpha(p_i) = y_i$ then it indicates that the degree of truth of proposition d_i is y_i. A transition t may fire if $\alpha(p_i) \geq \lambda$, $\forall \alpha(p_i) \in I(t)$, where λ is a threshold value between zero and one. When transition t fires, the token values of its input places are used to calculate the token values of its output places according to certain rule-types.

The basic reasoning rule is of form R_i : IF d_j THEN d_k (CF = μ_i) , where d_j and d_k are propositions. It can be modeled as shown in Fig.2, where propositions d_j and d_k are represented as the places p_j and p_k, respectively, and the causality between the two propositions is represented as the transition t_i. If $y_j > \lambda_i$, rule R_i becomes valid, and the degree of truth of the consequence proposition d_k is $y_j = y_i \times \mu_i$; otherwise, rule R_i is invalid and the transition t_i cannot be fired.

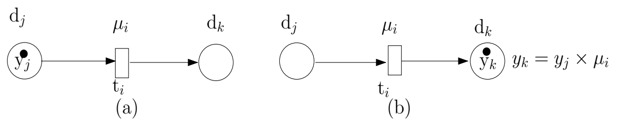


Fig. 2. The simplest FPN. (a)Before firing transition t_i. (b)After firing transition t_i.

If there exist some cases where the antecedent part or consequence part of a fuzzy rule contain "AND" or "OR" connectors, then it is called a composite fuzzy production rule. According to [13], composite fuzzy production rules are classified into the following three types:

– Type 1 : IF d_{j1} AND d_{j2} AND ... AND d_{jn} THEN d_k (CF =μ_i),where d_{jk} (1$\leq$ $k \leq n$) $\in D$. The fuzzy reasoning process of this rule can be modeled by an FPN as shown in Fig. 3. After fuzzy reasoning, the degree of truth of d_k is min(d_{j1} , d_{j2} , ... , d_{jn}) $\times$ μ_i.
– Type 2 : IF d_j THEN d_{k1} AND d_{k2} AND ... AND d_{kx} (CF =μ_i),where d_{kx} (1$\leq$ $x \leq n$) $\in D$. It can be modeled by a FPN as shown in Fig. 4.

– Type 3 : IF d_{j1} OR d_{j2} OR ... OR d_{jn} THEN d_k (CF $=\mu_i$),where d_{jk} ($1 \leq k \leq n$) $\in D$. The fuzzy reasoning process of this rule can be modeled by an FPN as shown in Fig. 5.

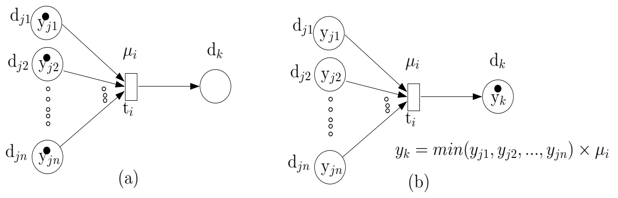


Fig. 3. Fuzzy reasoning process of Type 1 (a)Before firing transition t_i. (b)After firing transition t_i.

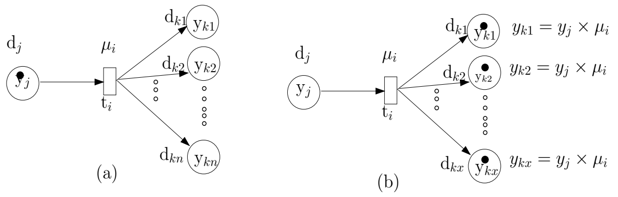


Fig. 4. Fuzzy reasoning process of Type 2. (a)Before firing transition t_i. (b)After firing transition t_i.

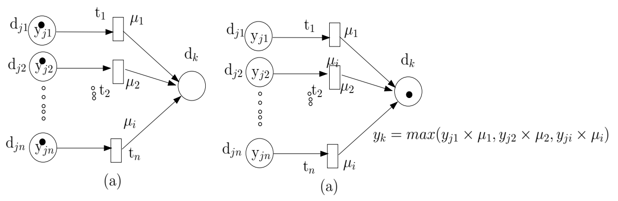


Fig. 5. Fuzzy reasoning process of Type 3. (a)Before firing transition t_i. (b)After firing transition t_i.

3.2 The Ontology Concept Similarity Value

The pure subsumption based approach has won great popularity in the ontology concept matchmaking. Typically, this approach tells us how similar one concept is to another by dividing the similarity into several concrete degrees, such as the approach in [4]. However, we are motivated to present an approach to calculate the similarity values over (0,1] instead of evaluating the similarity degrees such as exact, plugIn, subsumes, and fail. The similarity value can determine how similar one concept is to the other. By specifying the similarity value, the concept matchmaking procedure can be more effectively performed.

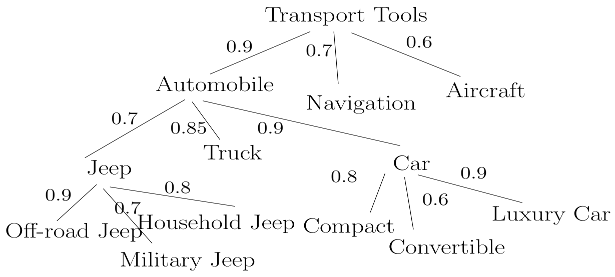


Fig. 6. The class hierarchy of transportation ontology

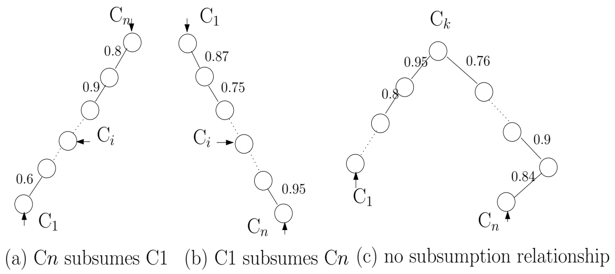


Fig. 7. The relation between two concepts

Given an ontology, a concept hierarchy, such as the one depicted in Fig. 6, can be derived. With each branch, a child concept is associated with a parent concept. Since the child concepts are derived by property restrictions, they are more concrete and can provide all the information available from their parent concepts. Thus, the similarity value is direction-dependent.

For two arbitrary concepts C_1 and C_n in the same ontology, their relationships are shown in Fig.7.

Let the similarity from two adjacent superclass C_1 to subclass C_2 be denoted by S(C_1, C_2). The value can be obtained according to [19].

SimValue(C_1, C_n) depending on their positions in the concept hierarchy can be calculated as follows.

- If C_1 and C_n are the same or C_1 is the subclass of C_n, as shown (a) in Fig.7, then *SimValue*(C_1, C_n)=1.
- If C_1 is an adjacent subclass of C_n, then *SimValue*(C_1, C_n)= S(C_1, C_n).
- If C_1 is a parent concept of C_n (n $\geq$ 2) as depicted in the position relationship (b) in Fig.7, then *SimValue*(C_1, C_n) is the product of the similarity values for a series of adjacent classes from C_1 class to C_n.
  SimValue(C_1, C_n)= *SimValue*(C_1, C_2) $\times$ *SimValue*(C_2, C_3) $\times \ldots \times$ *SimValue*(C_{n-1}, C_n)
- If there is a kind of relationship between C_1 and C_n as depicted in the position relationship (c) in Fig.7, a Least General Parent (LGP) concept C_k is calculated, where k$\geq$2 and n$\geq$k+1. C_k is the LGP concept of C_1 and

C_n if C_k is the parent concept of C_1 and C_n and there is no child concept of C_k that subsumes C_1 and C_n. In this case, the similarity value from C_1 to C_n is calculated as follows: $\mathit{SimValue}(C_1, C_n) = \mathit{SimValue}(C_k, C_1) \times \mathit{SimValue}(C_k, C_n)$

4 Transformation and Matchmaking

OWL-S is a semantic markup language that enables us to describe Web services so that available services can be selected, invoked and composed. The essential properties of a service are described by the following three classes: ServiceProfile, ServiceModel and ServiceGrounding. The ServiceProfile provides all the necessary information for a service to be found and possibly selected. In OWL-S, services are viewed as processes. So, the ServiceModel describes the service in terms of a process model. The ServiceGrounding defines how to access to the service by specifying the communication protocols and messages, and the port numbers to be used.

As part of the ServiceModel, a process model can be used to describe the interaction protocol between a service and its clients. An OWL-S process model is organized as a process-based workflow, and describes the behaviors of the web services. To determine whether two services can be matched, the OWL-S process models need to be transformed into FPNs. Process models can be grouped under two heads: atomic and composite processes. Thus, the atomic process is first modeled by the FPN that can describe four operation primitives. The composite processes are composed hierarchically of some processes using control constructs. Therefore, a composite process can be recursively transformed by unfolding subprocess until the FPN is only constructed by atomic processes and control constructs.

An FPN can be utilized to model a process model. Such a model is called an FPN process model.

Definition 1. An $FPN = (P, T, D, I, O, f, \alpha, \beta)$ is called an FPN process model if the following conditions can be satisfied:

(1) The set of places is divided into three disjoint sets: internal places P^N, input places P^I and output places P^O.

(2) The flow relation is divided into internal flow $F^N \subseteq (P^N \times T) \cup (T \times P^N)$ and communication flow $F^C \subseteq (P^I \times T) \cup (T \times P^O)$.

(3) The net $WN = (P^N, \mathrm{T}, F^N)$ is a workflow net.

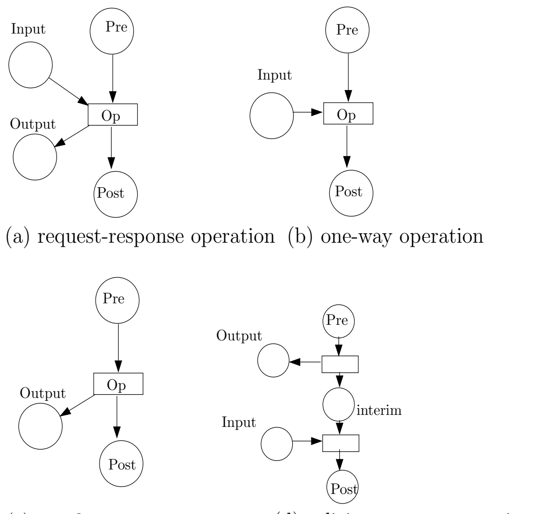
(4) $\mathrm{f(t)} = 1, \forall\ \mathrm{t} \in T$.

An FPN represents a composite process, where the net WN is the business protocol described by the composite process of the service;

4.1 Atomic Process Transformation and Operations Matching

Atomic processes correspond to operations that the service can be directly executed; they have no subprocesses. Each atomic process is described by four kinds of components: inputs, outputs, preconditions and effects. it can have several (includes zero) inputs, outputs, preconditions and effects. In general, the inputs and outputs of an atomic process can not be empty at the same time. For simplicity, only input and output are taken into consideration.

There are four web service operation primitives: request-response operation, one-way operation, notification operation, solicit-response operation, and the first three primitives can be described by OWL-S atomic processes. An atomic process with both inputs and outputs corresponds to a WSDL request-response operation, and can be represented by the FPN process model shown in Fig.8(a). The transition *Op* stands for the execution of an atomic process, place *Pre* represents the states before the execution of the atomic process, while place *Post* represents the states after the execution. Place *Input* is an input parameter, place *Output* is an output parameter. The places *Pre* and *Post* belong to P^N, while Places *Input* and *Output* belong to P^I and P^O respectively. An atomic process can have more than one input/output parameters that are represented by a group of places. The value of transition CF is 1. Similarly, an atomic process with inputs, but no outputs, corresponds to a WSDL one-way operation, as shown in Fig.8(b). An atomic process with outputs, but no inputs, corresponds to a WSDL notification operation, as shown in Fig.8(c).



(a) request-response operation (b) one-way operation

(c) notification operation (d) solicit-response operation

Fig. 8. Modeling four types of operation

OWL-S uses a composite process to describe solicit-response operation. It is a composite process with both outputs and inputs, and with the sending of outputs specified as coming before the reception of inputs. We model the solicit response operation with an FPN by combining the FPNs for the notification operation and the one-way operation correspondingly, as shown in Fig.8(d).

A process represents a communication activity. If two web services can interact with each other, the solicit-response operation and request response operation should be compatible. Similarly, the notification operation and one way operation should be compatible. Thus, we propose an algorithm written in pseudo code shown below to determine if the two operations can be matched.

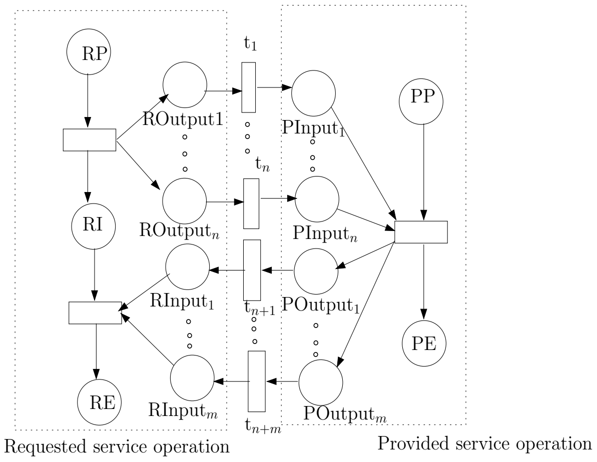


Fig. 9. Matchmaking of the two operations

Fig.9 shows the matchmaking of two operations. In the left dashed border, a solicit-response operation is described to represent a requested operation, and the set of output places $ROutputSet = \{\ ROutput_1, \ldots, ROutput_n\}$ corresponds to the operation's output parameters, while the input places corresponds to $RInputSet = \{\ RInput_1, \ldots, RInput_m\}$. Similarly in the right dashed border, a request response operation is described to represent provided operation.

The algorithm of matchmaking two operations

(1) Match the output parameters of the requested operation against the input parameters of the provided operation.

```
boolean RequestToProvidedMatch($ROutputSet$, $PInputSet$ , threshold)
1       x =| $ROutputSet$ | ; y=| PInputSet |;
2       if (y > x) return false;
3       for each $ROutput_i$ in $ROutputSet$, each $PInput_j$ in $PInputSet$
4           $\mu_i = Max(SimVal\ (Concept(ROutput_i), Concept(\ PInput_j)))$;
5           if ($\mu_i <$ threshhold) return false;
6           connect $ROutput_i$ to $PInput_j$ with $t_i$ whose Confidence Factor is $\mu_i$
7           $ROutputSet = ROutputSet$ - $ROutput_i$;
8           $PInputSet = PInputSet$ - $PInput_j$;
9       return true;
```

The input of the algorithm includes *ROutputSet* and *PInputSet*, which are sets of output and input places corresponding to the requested and provided service operations' parameters, respectively. The minimum similarity between *ROutputSet* and *PInputSet* is specified by the parameter threshold. At line 2, if the size of input sets of the provided service atomic operations is bigger than the size of output sets of the requested service operations, the two operations are not matched. At line 4, from each of *ROutputSet* and *PInputSet*, a place is picked out to form a place pair. Function *Concept* is used to get the concepts of the two places, while function *SimVal* and *Max* are applied to obtain the place pair which has the biggest concept similarity value. If the biggest similarity value is smaller than the threshold, the matchmaking fails; otherwise, the place $ROutput_i$ is connected to $PInput_j$ via t_i whose Confidence Factor is μ_i. Finally, $ROutpu_i$ and $PInput_j$ are removed from *ROutputSet* and *PInputSet* respectively. If *PInputSet* is empty, the for loop completes and the matchmaking is successful.

(2) Match the intput parameters of the requested operation against the output parameters of the provided operation.

boolean ProvidedToRequestMatch(*ROutputSet*, *PInputSet* , threshold)

This algorithm is quite similar to the algorithm in step 1. If the matchmaking is successful, and $RInput_i$ is connected to $POutput_j$ with t whose Confidence Factor is $Max(SimVal(Concept\ (POutput_j),\ Concept(RInput_i)))$.

(3) Check whether the two processes can be matched or not.

boolean RequestToProvidedMatch(ROutputSet, PInputSet, threshold)
$\cap$ boolean RequestToProvidedMatch(POutputSet, RInputSet, threshold)

This algorithm just evaluate the I/O matchmaking result. The two atomic processes can be matched if their I/O can be both matched.

4.2 Composite Process Transformation and Matchmaking

Composite processes are composed hierarchically of some processes using control constructs, which represent a series of message exchanges between the service and its users. The OWL-S process model provides nine control constructs : sequence, split, split+join, anyorder, if-then-else, choice, iteration, repeat-while and repeat-until, which specify the temporal sequences of the components.

Each composite process can be regarded as a tree structure, where the non-leaf nodes are control constructs and the leaf nodes are the atomic processes to be executed. Fig. 10 shows a tree that represents the OWL-S process model, and describes the behavior of the seller service in the motivation example. *SearchCarInfo*, *SendNotifyinStock*, *SendNotifyOutofStock*, *SetAccount*, *SetPinInfo* and *SendOrderConfirm* are six atomic processes that depict the communication activities between the seller service and the buyer service. They are composed hierarchically of atomic processes and composite processes. In Fig.10, *SellerService* process and *Continue* process are sequentially constructed and *ChoiceByStock* is of the Choice structure.

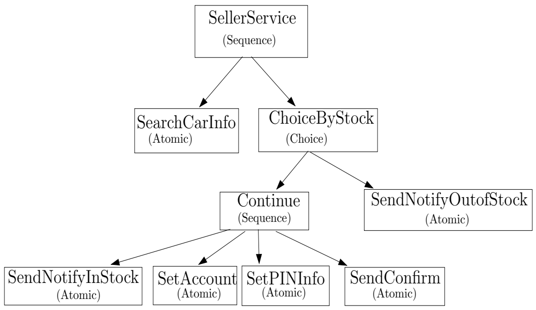


Fig. 10. The tree structure of the OWL-S process model

We transform the composite process into an FPN process modeled by the following steps:

(1) According to the business protocol of the web service, the composite process is transformed into a tree structure shown in Fig.10;

(2) The tree structure is transformed into a workflow net. We first define a transition to represent the web service. The transition just includes one input and one output place(representing preconditions and post-conditions respectively)as shown in Fig.11(a). Then this transition is refined using the sequence pattern. Two new transitions, namely *SearchCarInfo* and *ChoiceByStock*, are defined to replace the previous one. The two transition, *SearchCarInfo* represents an atomic process while *ChoiceByStock* represents a composite process as shown in Fig.11(b). Finally, it is transformed a workflow net as shown in Fig.11(d).

(3) According to the input/output parameters of the atomic process, the transitions of the workflow net are connected by directed arcs and places which represent the interfaces of the atomic process. Finally, an FPN process model in Fig.11(e) can be formed.

5 Combining FPN Process Modules

To evaluate the matching degree of the two semantic services, the FPN process models of the two services need to be combined. Then, the result FPN is analyzed and processed.

Let A = (P_a, T_a, D_a, I_a, O_a, f_a, α_a, β_a) and B = (P_b, T_b, D_b, I_b, O_b, f_b, α_b, β_b) be two FPN process models, they are combined into a composite FPN C = (P_c, T_c, D_c, I_c, O_c, f_c, α_a, β_a) with the following steps:

(1) Add two additional places *Cstart* and *Cfinish*, as well as two transitions t_α and t_β whose certification factor is 1. Then the starting and ending places of the two FPN process models are connected. Thus, $P_c = P_a \cup P_b \cup$ {Cstart, Cfinish }, $D_c = D_a \cup D_b \cup$ {"The matchmaking start", "The matchmaking finish"}, $T_c = T_a \cup T_b \cup \{t_\alpha, t_\beta\}$, $I_c = I_a \cup I_b \cup \{I_c(t_\alpha) = \{Cstart\}, I_c(t_\beta) = \{Afinish, Bfinish\}\}$ and $O_c = O_a \cup O_b \cup \{O_c(t_\alpha) = \{Astart, Bstart\}, O_c(t_\beta) = \{Cfinish\}\}$.

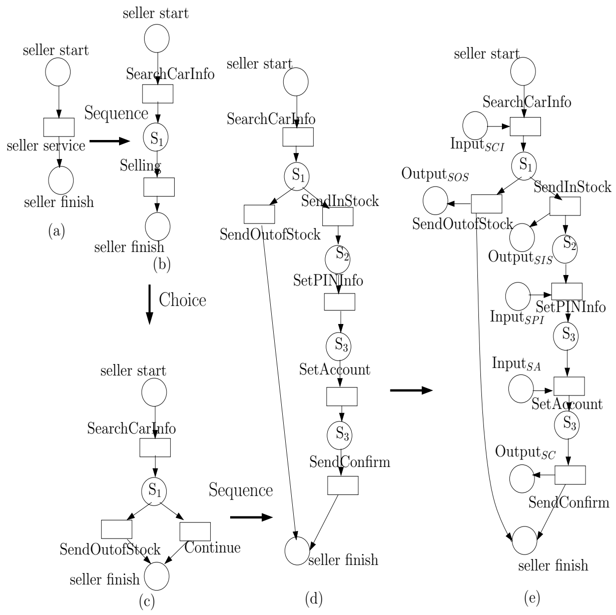


Fig. 11. The FPN composite process model transition

(2) Connect the input and output places of the matched two operations. Transition of the FPN process models represent an atomic process, and they can be composited into four different operation primitives to describe the requested operation and provided operation. We need to match each pair of requested operation and provided operations. If the requested operations in one composite process can be met by the other composite process, they are matched, otherwise they are not. Whether the two operations are matched can be determined by the algorithm introduced in Section 4. After the above steps, the places representing interfaces of operations can be connected by transitions with CF value. Then the two FPN process models can be combined into the composite FPN. Fig.12 shows the FPN of the matched operations in the motivation example.

6 Analysis

6.1 Analysis Soundness of FPN

The communication between two semantic web services is based on choreography. Thus, their behaviors must be compatible besides their interfaces are semantically equivalent. The composition of two FPN service models into a single FPN ensures that the semantics of their interfaces are equivalent. Hence, we need to verify deadlock-freeness of the composite FPN. A very efficient and widely accepted method to verify properties is the state space method [20]. The basic idea is to generate reachable states - as many as necessary to prove or

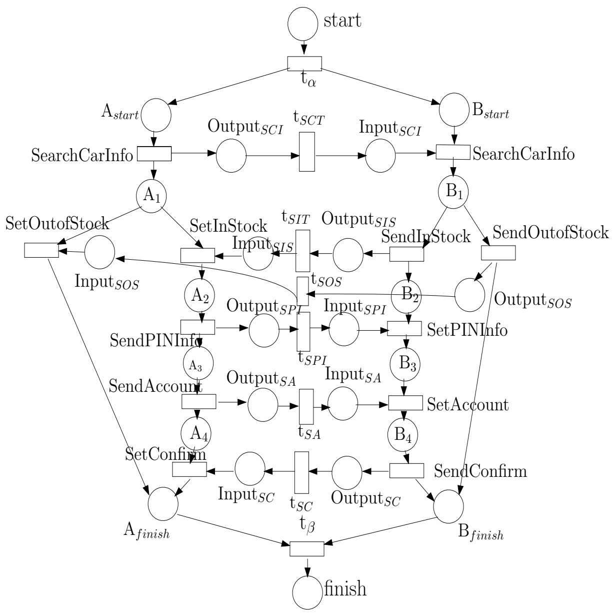


Fig. 12. Matchmaking of the two FPN process models

disprove the property. Each node of the graph represents one state of the FPN, and each arc represents the firing of one transition. Currently, many petri net tools verify the system soundness with this approach. When a deadlock is found, the tool generates the relevant part of the reachability graph. More details on the implemented reachability graph analysis can be found in [20].

In our motivation example, if the seller service requires the buyer service to provide account information before any further inquiry while the buyer service requires inquiries first, a deadlock happens. The communication between the two web services can be easily modeled by FPNs, and the trace from the initial state to the deadlock can be found through the state space approach. Thus, we draw the conclusion that our approach can effectively verify the behavior compatibility of two semantic web services.

6.2 The Matching Degree of Two Web Services

An FPN process model stands for the business protocol of a service, then the FPN resulted from the composition of the two FPN process models represents the interaction of the two services. Each place corresponds to a proposition, while each transition represents a fuzzy production rule. For example, the place *start* corresponds to the proposition: "the interaction starts"; the place end corresponds to the proposition: "the interaction end", the transition t_α represents the reasoning rule: IF "the interaction starts" THEN "the buyer service is ready" AND "the seller service is ready", with the Confidence Factor of the rule being 1;

Thus, the evaluation of the matching degree can be performed by an FPN reasoning process: first, the degree of truth of the initial places in the resulted FPN is 1, and according to the reasoning rules, the degree of truth of the goal place can be derived. The derived value represents the matching degree of the two services.

We use the FPN for the reasoning, since it has been proved to be a powerful representation method for the reasoning of a rule-based system. The popular reasoning algorithm can be classified into two types, forward reasoning[12,13,14] and backward reasoning [15,16]. The advantage of backward reasoning is that only information relative to the goal will be considered. This feature makes the reasoning process more flexible and more intelligent. Thus, we adopt the backward reasoning algorithm in this paper. First, an FPN is transformed into a Backward Tree according to the transformation rules; then, with the Backward tree, the reasoning process can be performed easily by firing the transitions from the top layer to the bottom layer; finally the degree of truth of the token in the goal place is exactly the reasoning result.

Illustration Example

For the sake of the simplicity, Fig.13 only shows a simplified interaction process of the buyer and seller services.

Let place *start*, *finish*, $P_1, P_2, \ldots, P_{12}$ be fourteen propositions.

D($start$): The interaction starts
D(P_1) : The buyer service starts
D(P_2) : The seller service starts
D(P_3) : The parameter provided by the buyer is A_α
D(P_4) : The parameter desired by the seller is A_β
D(P_5) : The buyer waits for inquiry result
D(P_6) : The seller prepares sending the inquiry result according to the stock
D(P_7) : The parameter desired by the buyer is B_α
D(P_9) : The parameter provided by the seller is B_β
D(P_8) : The parameter desired by the buyer is C_α
D(P_{10}) : The parameter provided by the seller is C_β
D(P_{11}) : The buyer service finish
D(P_{12}) : The seller service finish
D($finish$): The interaction finish

The interaction of the two web services can be regarded as a rule-based system. According to the previous interaction process, eleven fuzzy production rules can be derived, as well as the CF:

R_{t_α}: IF D(start) THEN D(P1) AND D(P2) (CF = 1)
R_{t_1}: IF D(P1) THEN D(P3) AND D(P5) (CF = 1)
$R_{t_{SCI}}$: IF D(P3) THEN D(P4) (CF = 0.95)
R_{t_2}: IF D(P2) AND D(P4) THEN D(P6) (CF = 1)
R_{t_3}: IF D(P5) AND D(P7) THEN D(P11) (CF = 1)
R_{t_4}: IF D(P5) AND D(P8) THEN D(P11) (CF = 1)
$R_{t_{SIS}}$: IF D(P9) THEN D(P8) (CF = 0.85)
R_{t_5}: IF D(P6) THEN D(P9) AND D(P12) (CF = 1)

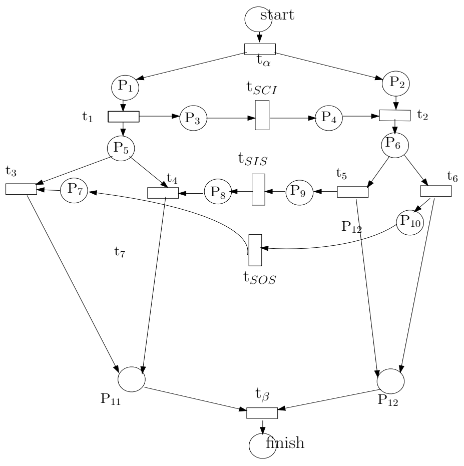


Fig. 13. A simplified interaction process of the buyer and seller services

R_{t_6}: IF D(P6) THEN D(P10) AND D(P12) (CF = 1)
$R_{t_{SOS}}$: IF D(P10) THEN D(P7) (CF = 0.70)
R_{t_β}: IF D(P11) AND D(P12) THEN D(finish) (CF = 1)

Let the degree of truth of the proposition D(start) be 1, then we want to know the degree of truth of proposition D(finish). According to the algorithm in [16], the FPN structure can be transformed into a Backward Tree, as shown in Fig.14.

After constructing the Backward Tree, transitions can fire in turn from the top layer (Layer 8) to the bottom layer (Layer 1). The reasoning process is shown as follows:

Layer 8: t_α fires, then KPS={{P_1, 1}}
Layer 7: t_1 fires, then KPS={{P_1, 1},{P_3, 1}}
Layer 6: t_α fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1}}
t_{SCI} fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95}}
Layer 5: t_2 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}}
t_1 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}}
Layer 4: t_α fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}}
t_6 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95} , {P_{10}, 0.95}}
t_5 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95} , {P_{10}, 0.95},{P_9, 0.95}
t_{SCI} fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}
,
{P_{10}, 0.95},{P_9, 0.95}}
Layer 3: t_1 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1}}
t_{SOS} fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66}}

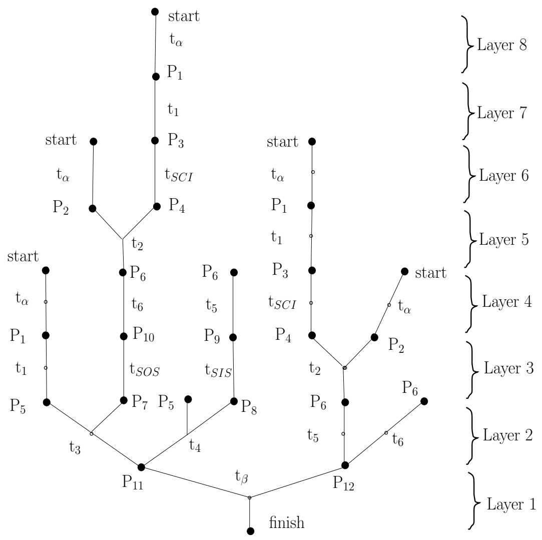


Fig. 14. Backward Tree of the FPN

t_{SIS} fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66},{P_8, 0.80}}

t_2 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66},{P_8, 0.80}}

Layer 2: t_2 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66},{P_8, 0.80}} {P_{11}, 0.66}}

t_4 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66},{P_8, 0.80}} {P_{11}, 0.80}}

t_5 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66},{P_8, 0.80}} {P_{11}, 0.80},{P_{12}, 0.95}}

t_6 fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66},{P_8, 0.80}} {P_{11}, 0.80},{P_{12}, 1}}

Layer 1: t_β fires, then KPS={{P_1, 1},{P_3, 1},{P_2, 1},{P_4, 0.95},{P_6, 0.95}, {P_{10}, 0.95},{P_9, 0.95},{P_5, 1},{P_7, 0.66},{P_8, 0.80}} {P_{11}, 0.80},{P_{12}, 1},{$finish$, 0.80}}

Now, the degree of truth of the goal place *finish* has been derived, and it is valued as 0.80. The value indicates the possibility of the interaction being successfully completed. It represents the matching degree between the two services.

7 Conclusion

To accurately and automatically discover required web services for dynamic and flexible integrating business processes, a formal approach by means of FPNs has been used. Our approach evaluates the compatibility of services based on the behavior of web services by semantic matching. It ensures that messages can be exchanged successfully between provided and requested services and produces expected effects as much as possible. FPNs are used to perform fuzzy reasoning and calculate the matching degree of two web services. Therefore, the most suitable services can be selected.

However, there is a limitation for using our approach to integrate business processes since we consider only a bilateral matching of web services in this paper. In the future, we will further investigate a multi-lateral matching of web services. Moreover, a prototype system needs to be developed to prove the feasibility of our approach.

References

1. Chinnic, R., Gudgin, M., Moreau, J-J., Weeawarana, S.: Web Services Description Language Version 1.2 (2003) http://www.w3.org/TR/2003/WD-wsdl12-20030303
2. Andrews, T., Curbera, F., Dholakia, H., et al.: Business Process Execution Language for Web Services (BPEL4WS) version 1.1 (May 2003)
3. The OWL-S Coalition. OWL-S 1.1 Draft Release (2004) http://www.daml.org/services/owl-s/1.1/
4. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.: Semantic Matching of Web Services Capabilities. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 333–347. Springer, Heidelberg (2002)
5. Martens, A.: Analyzing Web Service Based Business Processes. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 19–33. Springer, Heidelberg (2005)
6. Wombacher, A., Fankhauser, P., Mahleko, B.: Matchmaking for Business Processes based on Choreographies. In: Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (2004)
7. Lei, L., Duan, Z.: Transforming OWL-S Process Model into EDFA for Service Discovery. In: Proceedings of the IEEE international Conference on Web Services (Icws'06), vol. 00, pp. 137–144 (2006)
8. Jian, W.U.: Web service fuzzy matching in internet based manufacturing. Journal of Zhejiang University, vol. 9 (2006)
9. Xu, J., Zhu, Q., Li, J., Tang, J., Zhang, P., Wang, K.: Semantic Based Web Services Discovery. In: Proceedings of the AWCC 2004. LNCS, vol. 3309, pp. 388-393 (2004)
10. Sycara, K.: Dynamic Discovery, Invocation and Composition. In: Vouros, G.A., Panayiotopoulos, T. (eds.) SETN 2004. LNCS (LNAI), vol. 3025, Springer, Heidelberg (2004)
11. Moszkowski, B.C.: Reasoning about digital circuits. Ph.D Thesis, Stanford University. TRSTAN-CS-83-970 (1983)
12. Looney, C.G.: Fuzzy Petri nets for rule based decision making. IEEE Transactions on System, Man, and Cybernetics-Part A SMC218 (1), 178–183 (1988)
13. Chen, S.M., Ke, J.S., Chang, J.F.: Knowledge representation using fuzzy Petri nets. IEEE Transactions on Knowledge and Data Engineering 2(3), 311–319 (1990)

14. Pedrycz, W., Gomide, F.: A generalized fuzzy Petri net model. IEEE Transaction Fuzzy Systems 2, 295–301 (1994)
15. Chen, S M: Fuzzy backward reasoning using fuzzy Petri nets. In: IEEE Transactions SMC Part B: Cybernetics, vol. 30, pp. 846–855
16. Yang, R., Heng, P-A., Leung, K-S.: Backward Reasoning on Rule-Based System Modeled by Fuzzy Petri Nets through Backward Tree. FSKD, pp. 18-22 (2002)
17. Meimei, G., Mengchu, Z., Xiaoguang, H., et al.: Fuzzy reasoning Petri nets. IEEE Transactions on System, Man, and Cybernetics-Part A 33(3), 314–324 (2003)
18. Guangsheng, Z., Changjun, J., Zhijun, D.: Service Discovery Framework Using Fuzzy Petri Net. Journal of Computer Research and Development (2006)
19. Rong, Z.: Study on ontology based Web service discovery and composition technique: [Master dissertation] Shenyang: Northeastern University (2004)
20. Funk, K.: Petrinetz basierte Analyse von BPEL4WS Geschäftsprozessen. Master's thesis, Fachhochschule Nordostniedersachsen, FB Wirtschaft (2005)

Formal Models for Multicast Traffic in Network on Chip Architectures with Compositional High-Level Petri Nets

Elisabeth Pelz[1] and Dietmar Tutsch[2]

[1] LACL, Université Paris 12, Faculté de Sciences, Créteil, France
[2] Technische Universität Berlin, Real-Time Systems and Robotics, Berlin, Germany

Abstract. Systems on chip and multicore processors emerged for the last years. The required networks on chips can be realized by multistage interconnection networks (MIN). Prior to technical realizations, establishing and investigating formal models help to choose best adequate MIN architectures. This paper presents a Petri net semantics for modeling such MINs in case of multicast traffic. The new semantics is inspired by high-level versions of the Petri box algebra providing a method to formally represent concurrent communication systems in a fully compositional way. In our approach, a dedicated net class is formed, which leads to three kinds of basic nets describing a switching element, a packet generator, and a packet flush. With these basic nets, models of MINs of arbitrary crossbar size can be established compositionally following their inductive definition. Particular token generation within these high-level nets, as for instance, random load, yields an alternative approach to the use of stochastic Petri nets as in previous studies. The simulation of the models under step semantics provides a basis for performance evaluation and comparison of various MIN architectures and their usability for networks on chips. Particularly, multicast traffic patterns, which are important for multicore processors, can be handled by the new model.

1 Introduction

For the last years, systems on chip (SoC) emerged to build entire systems on a single chip. It is a result of the ongoing improvements in component integration on chips. For the same reason, multicore processors attracted interest of the processor developers.

To allow cooperating cores on such a multicore processor chip, an appropriate communication structure between them must be provided. In case of a low number of cores (e.g. a dual core processor), a shared bus may be sufficient. But in the future, hundreds or even thousands of cores will collaborate on a single chip. Then, more advanced network topologies will be needed. Many topologies are proposed for these so called networks-on-chips (NoCs) [1,4,6,7,17]. Multistage interconnection networks (MINs) could be an option to realize NoCs [9,16] and will be considered in this paper.

Some investigations on MINs as networks-on-chips already exist: For instance, to map the communication demands of on-chip cores onto predefined topologies like MINs, but also torus, etc., Bertozzi et al. [6] invented a tool called NetChip (consisting of SUNMAP and xpipes). This tool provides complete synthesis flows for NoC architectures. Another example where MINs serve as NoC is given by Guerrier and Greiner [9] who established a fat tree structure using Field Programmable Gate Arrays (FPGAs).

They called this on-chip network with particular router design and communication protocol Scalable, Programmable, Integrated Network (SPIN). For different network buffer sizes the performances were compared with this tool. Alderighi et al. [1] used MINs with the Clos structure. Multiple parallel Clos networks connect the inputs and outputs to achieve fault tolerance abilities. Again, FPGAs serve as basis for realization.

But previous papers only considered unicast traffic in the NoC. It is obvious that multicore processors also have to deal with multicast traffic. For instance, if a core changes a shared variable that is also stored in the cache of other cores, multicasting the new value to the other cores keeps them up to date. Thus, multicast traffic constitutes a non-negligible part of the traffic. As consequence, networks for multicore systems should outstandingly support multicast traffic.

To choose the best adapted MIN topology for multicast traffic of multicore processors, it is necessary to study the numerous topologies widely. As Jantsch [10] stressed in his last year lecture, the need for formal models is very important, to allow simulations and performance analysis preliminarily to technical realizations. Jantsch himself established an abstract model of computation (MoC) for performance evaluation of NoCs, which was applied to a mesh topology.

Such formal models like the MoC currently exist only for few network topologies. Our work aims to contribute to this area by investigating MINs. The first originality of our work consists of proposing a general modeling concept which allows to consider networks of arbitrary size, which are built out of components, i.e. switching elements, of arbitrary size. The second originality is to treat stochastic events in a non stochastic model, in which just arbitrary produced data will be processed and which is easier to analyze. This overcomes the drawback of ordinary mathematical modeling concepts, i.e., the huge model development time since complex systems of equations have to be established (see, for instance, [14,19,21] for mathematical models of MINs). Moreover, simulation can be easily applied to our Petri net model. A model as simple as possible helps either to accelerate simulation run time or to keep the mathematical model small enough to be manageable and solvable by a computer tool.

This paper presents a Petri net semantics for modeling multistage interconnection networks in the case of multicast traffic in a very generic way. Up to now, Petri net models of MINs for multicast traffic were only proposed for a fixed crossbar size c, usually $c = 2$ [15]. Petri net models of MINs with arbitrary crossbar size exist only for unicast traffic, for instance [2]. But Bashirov and Crespi [2] only investigated the permutation capability of MINs operating in circuit switching mode. Our model determines performance results like throughput and delay times in case of packet switching and multicast traffic. Crossbars of arbitrary size c can be handled. This becomes possible by a fully compositional approach inspired by high-level (i.e. colored) versions of the Petri box algebra providing a method to formally represent concurrent communication systems. A dedicated net class is proposed, which leads to three kinds of basic nets describing a switching element, a packet generator and a packet flush. With these basic nets, models of MINs can be established compositionally following their inductive definition. The use of high level nets, instead of stochastic Petri nets, as for instance in our previous studies [15], makes simulation and analysis really easier and does not need specialized tools. For instance, a generalized step semantics reveals to be the adequate execution mode of this model.

Our approach has already been applied for unicast traffic [13]. But considering multicast traffic needs a more sophisticated net semantics. Such a net semantics for dealing with multicasting is given in this paper.

The paper is organized as follows. Section 2 describes multistage interconnection networks and the concept of multicast. The considered net class and the basic nets are introduced in Section 3 assuming that the Petri net formalism is well-known to the reader. Section 4 defines the formal Petri net semantics of arbitrary MINs. The simulation and evaluation of such models is presented in Section 5. Finally, a conclusion and ideas for future works can be found in Section 6.

2 Multistage Interconnection Networks

Multistage Interconnection Networks (MINs) are dynamic networks which are based on *switching elements* (SE). The most common approach to SEs are crossbars. SEs are arranged in *stages* and connected by *interstage links*. The link structure and amount of SEs characterizes the MIN.

MINs [19,21] with the *banyan property* are networks where a unique path exists from an input to an output. Such MINs of size $N \times N$ (N inputs and N outputs numbered from 0 to $N-1$, respectively) consist of $c \times c$ switching elements (SEs of c inputs and c outputs numbered from 0 to $c-1$, respectively) with $N = c^n$ and $n, c, N \in \mathbb{N}$. The number of stages is given by n. Figure 1 depicts a 3-stage MIN, with detailed numbering of one 2×2 SE, the bottom left one of the MIN.

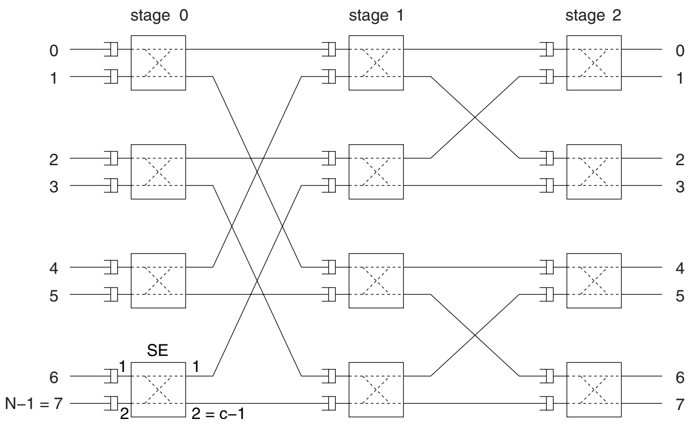


Fig. 1. 3-stage MIN consisting of 2×2 SEs

A message to be transferred in the MIN is divided into *packets* of equal size. The packets consist of a *header* representing the routing tag and of a *payload field* filled with the corresponding part of the message. To allow multicasting, the *routing tag* is

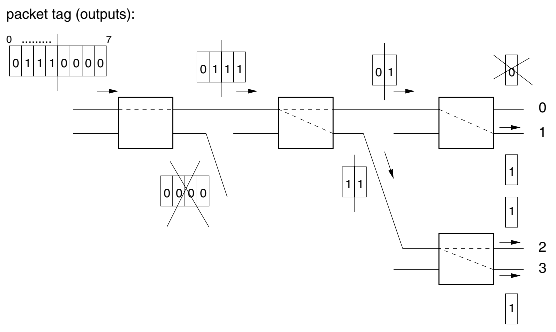


Fig. 2. Multicasting by tag

given as a binary N-digit number, each digit representing the corresponding network output. The digits of the desired outputs for the packet are set to “1”, all others to “0”. Routing is realized as follows. If the packet arrives at a $c \times c$ SE, the tag is divided into c subtags of equal size. Each subtag belongs to one switch output, the first (lower indices) subtag to the first output, the second subtag to the second output, etc. If a subtag contains at least one “1”, a copy of the packet is sent to the corresponding output containing the subtag as the new tag. Figure 2 gives an example. A part of an 8×8 MIN consisting of 2×2 SEs is shown. A packet, which is destined to Output 1, 2, and 3, resulting in tag 01110000, crosses the network. At each stage, the tag is divided in the middle into two ($c = 2$) subtags. According to the existence of at least one “1” value in the subtag, a copy of the packet is sent to the corresponding SE output.

To achieve synchronously operating switches, the network is internally clocked by a central clock. Because packet switching is applied, *buffers* can be introduced. In each stage k ($k \in \mathbb{N}$ and $0 \leq k \leq n - 1$), there is a FIFO buffer of equal size in front of each SE input to store a maximum number of m_{max} packets ($m_{max} \in \mathbb{N}$). The packets are forwarded by *store-and-forward switching* from a stage to its succeeding one: at most, a single stage per clock cycle is passed to keep the hardware amount for switching reasonable small. Due to the synchronously operating switches, all packets in all stages that are able to move are forwarded simultaneously to the next stage.

Packets that are destined to full buffers can be handled by dropping those packets [20] or by applying *backpressure* mechanism. The backpressure mechanism keeps packets in their current stage until the required buffer becomes available again. That means that no packets are lost within the network in contrast to the method of dropping packets. Local and global backpressure are distinguished. *Local backpressure* only observes the destination buffer of the next stage: The packet of stage k is sent if space at stage $k + 1$ is available. *Global backpressure*, which we will consider in the sequel, gets additional information about packet flows: the packet of stage k is sent even if no space at stage $k + 1$ is currently available but becomes available till the packet is received. Such a situation may arise if a packet leaves stage $k + 1$ at the same clock cycle. Global backpressure impresses by a higher performance.

If a packet is destined to multiple buffers of the succeeding stage (multicast) and some of the destination buffers are not available, copies of the packet are only sent to the available buffers. But the packet stays in its current buffer till all buffers for which it is destined have been reached. Such a behavior is called *partial multicast* and is the one considered in this paper.

Numerous multistage interconnection network topologies are given in the literature [12]. The structure of their *interstage connections* distinguishes them. They are referred to as Omega, Flip, Baseline, Indirect Binary Cube (IBC), and Modified Data Manipulator (MDM). Some of them can be defined by induction, which means that a MIN with n stages is built from MINs with at most $n-1$ stages, as for instance, the $c^n \times c^n$ *Modified Data Manipulator*, named $\mathsf{MDM}(c, n, m_{max})$ where the parameters c and m_{max} need to be fixed. The MDM topology will be investigated in the rest of the paper, but we could have chosen any other inductively defined MIN as well.

MDM construction

First step $i = 1$: The smallest MIN of this type is given by a $\mathsf{MDM}(c, 1, m_{max})$ network, which consists in a single $c \times c$ SE.

Now, it is assumed that all MINs $\mathsf{MDM}(c, j, m_{max})$ with $1 \leq j \leq i$ already exist.

Step $i + 1$: MIN $\mathsf{MDM}(c, i + 1, m_{max})$ can be set up as follows (see Figure 3 as an example with $c = 2$ and $i = 2$): Stage 0 is built by placing c^i SEs in a column. The

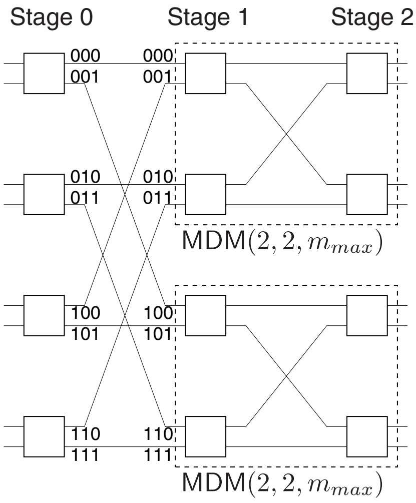


Fig. 3. Establishing $\mathsf{MDM}(2, 3, m_{max})$ out of $\mathsf{MDM}(2, 2, m_{max})$

inputs as well as the outputs of this SE column are c-ary numbered from top to bottom, respectively. These numberings start with 0 and end with $c^{i+1} - 1$ each. Thus, a c-ary i-digit number $\alpha_i \alpha_{i-1} \alpha_{i-2} \ldots \alpha_2 \alpha_1 \alpha_0$ results for each input and output, respectively. Such a number can be factorized as $\alpha_i \beta \alpha_0$ where β includes all digits except the first and last one (cf. Section 4).

The remaining Stages 1 to i of the $c^{i+1} \times c^{i+1}$ MIN are established by placing c copies of $c^i \times c^i$ MINs in a column. The inputs at the left side of this column of MINs are also c-ary numbered from 0 to $c^{i+1} - 1$ from top to bottom. The same numbering is applied to the outputs at the right side of the column of MINs.

To finish the construction of the $c^{i+1} \times c^{i+1}$ MIN, the *interstage connections* between Stage 0 and Stage 1 must be established according to the following *MDM rule*: the input $\alpha_i \beta \alpha_0$ of the column of $c^i \times c^i$ MINs is connected to the output $\alpha_0 \beta \alpha_i$ of the SEs at Stage 0. That means the corresponding output number is found by simply exchanging the first and last digit of the input number.

3 Compositional High-Level Petri Nets

As explained in the introduction, the model used in our approach is inspired by high-level versions of the Petri box algebra [3,5,11] which allow to formally represent concurrent communicating systems and to study various properties of their executions. These nets $\mathsf{N} = (P(\mathsf{N}), T(\mathsf{N}), A, \gamma)$, like other high-level Petri net models, carry the usual annotations, given by a labeling γ on places $P(\mathsf{N})$ (types, i.e., sets of allowed tokens), on transitions $T(\mathsf{N})$ (guards which are Boolean expressions, γ playing the role of occurrence conditions) and on arcs A (multisets of net variables).

3.1 A Dedicated Small Net Class

To model MINs, we may restrict us to a quite small, particular high-level net class where an arc inscription always consists of exactly one variable and where two kind of places exist: *buffer places* which all have the same type (lists of numbers representing queues of packet headers in the buffers of SEs) and *entry places* which have trivial black token type. From the set of operations, which give all box-like models an algebraic structure, we only need two: renaming and parallel composition.

The *marking* of such a net associates to each place a unique value (token) from the type of the places. If the net N is given with an *initial marking* M_0 we call it a *net system* $\Sigma = (\mathsf{N}, M_0)$. The *transition rule* is like for other high-level nets; namely, a transition t can be executed at a given marking if the inscriptions of its input arcs can be mapped to values which are present in the input places of t and if the guard of t, $\gamma(t)$, evaluates to true under that mapping, more precisely, under an extension of it to the variables on the output arcs. The execution of t under this mapping transforms the marking by removing values from the input places of t and by depositing values in the output places of t (accordingly to the chosen mapping).

Note that given a transition t and a marking M, there may be, in general, several possible executions (firings) depending on different mappings of variables to values (called *bindings* and denoted by σ) in the inscriptions around t. But as all places are always marked with exactly one token (cf. Proposition 4 in a later section) the binding σ will be fixed on the restriction to variables on the input arcs of t. Only if the guard allows mappings of the variables on the output arcs to several values, there may exist several total bindings (corresponding to a non-deterministic choice). We write $M((t : \sigma) > M'$ if the firing of t under binding σ at marking M yields the new marking M'. We call *reachable markings* all markings which can be reached from the initial one after firing a finite sequence of transitions.

Now we distinguish precisely the following sets of tokens as types for the places in our nets. The type $\{\bullet\}$ will be that one for entry places.

We will have a parameterized type $BList(c, m_{max})$ (for **B**inary **List**s) if dealing with nets for $c{\times}c$ SEs with buffer size m_{max} for MINs : its elements are strings (lists) of at most m_{max} binary numbers, each of length c^k for some $k > 0$, seen as c blocks of binary numbers of same length c^{k-1}. These token represent a queue of packet headers (like this one in Figure 2) in buffers with maximal length m_{max} . This type may be restricted to numbers (initial addresses) of a given length c^n, called type $BList_n(c, m_{max})$. The smallest token of binary list type, ε, will represent the empty list, and **0** a block of zeros.

As operations on this type, we only allow the following ones, where x and y being lists:

- $x[0]$ for taking the head of the list x, returning ε if list x is empty.
- $x[0]^h$ for reading/taking the h-th block of the head of the list x.
- $x[1:]$ for taking the tail of x, returning ε if list $x[1:]$ is empty.
- $\prec_{max}(x)$ the predicate being true if the length of list x is smaller than m_{max}.
- $nb[x]$ the predicate being true if the length of list x is one, i.e., if the list x contains exactly one number c^k.
- $x + y$ for appending y to list x
- For z being an already defined number or block,
  $z = \mathbf{0}$ for testing if z is a block of zeros, i.e. if z did not contain some 1.

Example: Let us consider the following list x from $BList(c, m_{max})$ with $c = 2$ and $m_{max} = 10$, with its head, tail and some blocks indicated. Here $k = 3$, so each binary number has length eight. The list consists of four numbers, so its length 4 is less than m_{max}:

$$x = \overbrace{\underbrace{0111}_{x[0]^0}\underbrace{0000}_{x[0]^1}}^{x[0]}.\overbrace{10101000.00100000.11000100}^{x[1:]}$$

All of the following conditions are true for the given example: $\prec_{max}(x)$, $x[0] \neq \mathbf{0}$, $x[0]^0 \neq \mathbf{0}$, $x[0]^1 = \mathbf{0}$, $nb[\,x[0]\,]$ and $\neg\, nb[\,x[1:]\,]$. Note that only the header of the SE buffers are represented by our token type for buffer places. The payload field can and will be neglected, to avoid heavy structures and because this data part is not important for the modeling of routing and consecutive performance evaluation.

3.2 Net Composition Operations

A *renaming* consists in a function denoted by a Greek letter, for instance $\nu(\mathsf{N})$, $\mu(\mathsf{N})$ or $\rho(\mathsf{N})$, renaming the places and transitions of a net N. They are explicitely defined on the part of the domain, where names have to be changed, and are id (i.e., stay identical) everywhere else.

The *parallel composition*, denoted by $\mathsf{N}_1 \| \mathsf{N}_2$, consists of putting N_1 and N_2 side by side while buffer places $p_1 \in \mathsf{N}_1$ and $p_2 \in \mathsf{N}_2$ having the same name are merged. If one of the merged places has the restricted binary list type $BList_n(c, m_{max})$ for a given n, the resulting place will also have it.

Note that renaming will be used for N_1 and/or N_2 in order to ensure that places which should be merged have the same name.

3.3 Basic Net $\mathsf{N}_{SE}(c, m_{max})$

By analogy with SEs, which are the basic components of the MINs, we introduce a basic net modeling one SE. We also propose basic nets which are able to represent the packet generators and receivers in a later section. Each of them has a single transition, and some input/output places around.

Let us describe first the basic nets for SEs, simply called $\mathsf{N}_{SE}(c, m_{max})$. They depend on two parameters, c and m_{max} for dealing with $c \times c$ SEs with maximal buffer size m_{max}. $\mathsf{N}_{SE}(c, m_{max})$ consists of a single transition t and $2c$ buffer places of type $BList(c, m_{max})$, which are all connected with one input and one output arc to t, cf. Figure 4. The c "left places", representing the SE input buffers, are *named* from top to bottom ℓ_0 to ℓ_{c-1}. Similarly, the c "right places", covering the SE outputs, are *named* r_0 to r_{c-1}. The arc inscriptions are as shown in Figure 4 below, chosen in such a way that they also express the direction of the flow: if the arc enters the transition we have x_i and y_i, if it leaves it, the same names are primed.

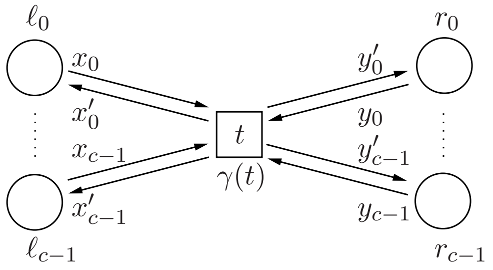


Fig. 4. Basic net $\mathsf{N}_{SE}(c, m_{max})$ for a $c \times c$ SE

Role of the guard

The guard $\gamma(t)$ determines when and how the transition fires. We require from the global net that we will build later on, the property that all reachable markings are such that each place contains exactly one token. This will be proved later (cf. Proposition 3 in Section 5.2).

The condition of the firing in the language of the SEs performing the backpressure mechanism is to investigate each block h of every first header. If it isn't zero, it becomes an address of the destination output, under the condition that the h-th output buffer is not full. The effect is to shift then the h-th block of a header to the h-th SE output, if it is not full, and not moving it, if this output is full or if a shift from a concurrent input to this output is executed.

Translated in the net language the condition will be determined by the tokens around t: each block $h \in \{0, c-1\}$ of the first number in each input place will tell us if a transfer to the h-th output is asked for, and each token in an output place tells us if it reaches maximum size (the place is "full") or not (the place stays "available").

Here we need to express this by logical formulae (without quantifiers, i.e. boolean expressions), built over variables, constants (values) of type $BList(c, m_{max})$ and the operations of this type.

Construction of the guard

We will use the following convention to improve readability: inscriptions of arcs on the left side of the transition, i.e. x or x', will always have indices $\mathbb{I}, i, i', \ldots$, and those of arcs right to the transition, i.e. y or y', will have indices $\mathbb{J}, \mathbb{H}, j, j', \ldots$ or h, h'. The following auxiliary notations need to be introduced, where $\mathbb{C} = \{0, \ldots, c-1\}$ is the c-ary alphabet.

For a set of pairs $\mathbb{S} \subseteq \mathbb{C}^2$, its $\mathbb{I}$ and $\mathbb{H}$ -sides, i.e. its first and second projections, are

$$\begin{array}{lcl} \mathbb{I}(\mathbb{S}) & = (\ \mathbb{S}\restriction_1\ ) = & \{i \in \mathbb{C} \mid \exists h\ (i,h) \in \mathbb{S}\,\} \\ \mathbb{H}(\mathbb{S}) & = (\ \mathbb{S}\restriction_2\ ) = & \{h \in \mathbb{C} \mid \exists i\ (i,h) \in \mathbb{S}\,\}. \end{array}$$

For an element $i \in \mathbb{C}$, $H(i) = \{h \in \mathbb{C} \mid \ (i,h) \in \mathbb{S}\,\}$ defines the right sides of pairs in $\mathbb{S}$ with left side i.

The guard is given by the following formula, whose intuitive meaning -line by line- will become clear in the proof of its satisfiability under every marking (see next proposition).

$$\begin{array}{l} \gamma_{MC}(t) \ \equiv \ \bigvee_{\mathbb{J}\subset\mathbb{C}} \Big(\bigwedge_{j\in\mathbb{J}} \prec_{max}(y_j) \ \wedge \bigwedge_{j\in\mathbb{C}\setminus\mathbb{J}} \big(\neg \prec_{max}(y_j) \ \wedge \ y'_j = y_j\big) \ \wedge \\ \Big[\bigvee_{\mathbb{S}\subset\mathbb{C}^2 \text{ with } |\mathbb{S}|=|\mathbb{H}(\mathbb{S})|\ ,\ \mathbb{H}(\mathbb{S})\subseteq\mathbb{J}} \Big[\Big(\bigwedge_{j\in\mathbb{J}\setminus\mathbb{H}(\mathbb{S})} \big(y_j = y'_j \wedge \bigwedge_{i\in\mathbb{C}} x_i[0]^j = \mathbf{0} \big) \Big) \\ \wedge \Big(\bigwedge_{i\in\mathbb{I}(\mathbb{S})} \Big[\bigwedge_{h\in H(i)} (\, x_i[0]^h \neq \mathbf{0} \wedge y'_h = y_h + x_i[0]^h \,) \Big] \\ \wedge \ \Big[\Big(\bigwedge_{h\notin H(i)} x_i[0]^h = \mathbf{0} \ \longrightarrow x'_i = x_i[1:] \Big) \vee \\ \Big(\bigvee_{h\notin H(i)} x_i[0]^h \neq \mathbf{0} \ \longrightarrow \Big[\bigwedge_{h\notin H(i)} x'_i[0]^h = x_i[0]^h \ \wedge \ x'_i[1:] = x_i[1:] \\ \wedge\, (\bigwedge_{h\in H(i)} x'_i[0]^h = \mathbf{0}\,) \Big] \Big) \Big] \Big) \ \wedge \ \bigwedge_{i\notin\mathbb{I}(\mathbb{S})} x'_i = x_i \ \Big] \Big] \Big) \end{array}$$

Proposition 1: Consider an arbitrary marking of $\mathsf{N}_{SE}(c, m_{max})$ verifying that each place contains exactly one token. Then, the transition is firable, i.e. there is at least one binding under which the guard is true.

Proof: First let us bind all x_i and y_j to the unique tokens present in the corresponding places. Let us show how to find an appropriate extension of this partial binding under which the formula becomes true.

Because each right place is marked, there is **exactly one** set of indices $\mathbb{J}$ for which the tokens in these places do not have the maximal size (i.e., the buffers there are still available) and the tokens in the others ones have the maximal size (the buffers there are full). Thus exactly one subformula of the outer disjunction can be true.

First case: If in the disjunction $\mathbb{J} = \emptyset$ then the formula becomes

$$\left(\bigwedge_{j \in \mathbb{C}} \left(\neg \prec_{max} (y_j) \ \wedge y'_j = y_j \right) \wedge \ \bigwedge_{i \in \mathbb{C}} x'_i = x_i \right)$$

expressing the situation where nothing is changed by the firing of t, i.e., the marking does not change, which is consistent with the fact that all output buffers are full.

General case: $\mathbb{J}$ is not empty. We have to show, that for this $\mathbb{J}$ the formula in big parenthesis is true:

Now we can always choose one $\mathbb{S}_0$ (between the $\mathbb{S}$ satisfying the big conjunction of Line 2, being a **maximal choice** of pairs (i, j) with $j \in \mathbb{J}$, thus $\mathbb{H}(\mathbb{S}_0) \subseteq \mathbb{J}$, expressing the movements to be done of a packet header from SE input i to SE output j. Each output j can be served by only one i, expressed by the condition $\mid \mathbb{S}_0 \mid = \mid \mathbb{H}(\mathbb{S}_0) \mid$. But each input $i \in \mathbb{I}(\mathbb{S}_0)$ can have several considered outputs $j \in H(i)$ due to multicast.

The **maximality** of the set $\mathbb{S}_0$ is expressed in the conjunction of Line 2: for each available j not included in $\mathbb{H}(\mathbb{S}_0)$ no input i desires a transfer to this j (i.e., all i-th subtags are **0**), and the token in place r_j does not change (i.e. $y'_j = y_j$).

It remains to show that the big conjunction of Lines 3 to 6 insures correctly the **transfer** from each concerned input $i \in \mathbb{I}(\mathbb{S}_0)$:

- Line 3 tells that the first routing tag in ℓ_i asks the $h \in H(i)$ for outputs, i.e. $x_i[0]^h \neq$ **0** , and each corresponding transfer of the packet header is done by appending the h-th subtag to the token in r_h (by $y'_h = y_h + x_i[0]^h$).
- Now two cases are to be distinguished, the disjunction between the subformula of line 4 and that of lines 5-6 tells this:
  - for no other h a transfer is asked for, nothing else needs to be done and the whole first tag in the buffer (token) can be omitted, by taking the tail of the list as new token (Line 4).
  - there are desired output addresses h, which could not be considered in the choice of $\mathbb{S}_0$ (i.e. $(i, h) \notin \mathbb{S}_0$), they need to be treated at a later moment, thus the token x_i in ℓ_i stays the same (Line 5) except for the subtags for which the transfer was done (Line 6, left side).
- Finally, the tokens in the other input places will stay the same, i.e., $x'_i = x_i$ (right conjunction of Line 6).

Now we can take values for the x'_i and y'_j according to the transfers fixed in the set $\mathbb{S}_0$ just described above, which completes the binding.

3.4 Basic Net $\mathsf{PG}(c, m_{max})$

In order to observe the behavior of the net models of MINs, which we will obtain later, we need some way to produce test data for simulation. Thus we will introduce a second kind of basic net, that one for a packet generator, depending on the same parameters, and called $\mathsf{PG}(c, m_{max})$. It consists of one transition t' linked to one entry place p_e of type black token by a loop (arc inscription may be omitted). There is also one buffer place ℓ of type $BList_n(c, m_{max})$ for some n, i.e. we fix here the length of the numbers to that of the desired addresses. It will become an interface with a net model of MINs, with an output arc inscribed by y and an input arc inscribed by y', cf. Figure 5(a). Note that place p_e becomes a run place once it is marked by one $\bullet$.

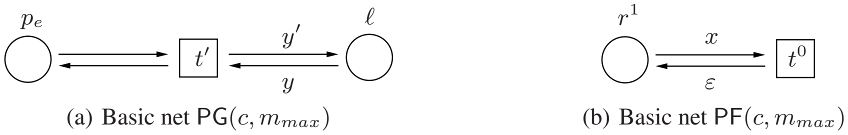

p_e
t'
y'
y
ℓ
(a) Basic net $\mathsf{PG}(c, m_{max})$
r^1
x
ε
t^0
(b) Basic net $\mathsf{PF}(c, m_{max})$

Fig. 5. Basic nets for packet generation and packet flush

Choice of a guard

We have different options to define the guard $\gamma(t')$ depending on which network load hypothesis we would like to feed later on the simulations. We will quote some of the possibilities, standard and non standard ones, already assuming that the token in place ℓ is initially the empty list ε:

(1) each firing adds just one arbitrary address to the token in ℓ if possible[1], corresponding to a very regular but permanent load:

$$\gamma_1(t') \equiv (\ y' = y + z \wedge nb[z]\ )\ \vee\ (\ y' = y\ \wedge \neg \prec_{max} (y)\ )$$

(2) each firing adds just no (ε) or one arbitrary address to the token in ℓ corresponding to a nondeterministic small load

$$\gamma_2(t') \equiv (\ y' = y + z\ \wedge\ (nb[z]\ \vee\ z = \varepsilon\ )\ )$$

(3) each firing fills the buffer to its maximal size representing very high traffic load

$$\gamma_3(t') \equiv (\ y' = y + z\ \vee\ y' = y\ )\ \wedge \neg \prec_{max} (y')\ )$$

(4) always the same address, for instance a_0, is added to the token in ℓ if possible[1], to investigate traffic streams

$$\gamma_4(t') \equiv (\ y' = y + a_0)\ \vee\ (\ y' = y\ \wedge \neg \prec_{max} (y)\ )$$

(5) each firing appends no or several arbitrary addresses x to the token already in ℓ, which express a totally random load

$$\gamma_5(t') \equiv y' = y + z$$

Note that the length of the value of y' is always less or equal to m_{max}, due to the type of the buffer place ℓ. Thus never a too long z can be appended to y, for instance by choice (2) or (5).

[1] I.e. for the case the token in place ℓ has already maximal size, we will always put the alternative $y = y'$ in the guard.

Notice that this basic net is constructed in a such way, that its only transition is always firable:

Proposition 2: Consider the net $\mathsf{PG}(c, m_{max})$ and $\gamma(t')$ one of the five guards defined as above. From each marking of this net with one black token in place p_e and one token of buffer type in place ℓ, the transition t' is firable.

Proof: Trivial, due to the construction of the net and the guard $\gamma(t')$, which can always be set true by an appropriate binding, for instance by choosing the value ε for z in the last choice $\gamma_5(t')$.

A network traffic, like (1), (3) or (4), should be selected in order to study worst case throughput. If nothing else is specified, we assume in the sequel that $\gamma(t') = \gamma_5(t')$, which seems to be a quite realistic choice.

3.5 Basic Net $\mathsf{PF}(c, m_{max})$

Finally, to represent the arrival or reception of a packet at the output of the MIN, to which it was destined, we need a way to ”empty” (i.e., to set to ε) a given buffer place. We call such a net packet flush $\mathsf{PF}(c, m_{max})$, given in Figure 5(b), where the buffer place r^1 is of type $BList(c, m_{max})$. The firing of transition t^0, with trivial guard ”true”, always sets the token in the place r^1 to ε.

The following proposition is trivial, as the trivial guard annotates t^0:

Proposition 3: From each marking of the net $\mathsf{PF}(c, m_{max})$ with one token of buffer type in place r^1, the transition t^0 is firable.

4 Net Semantics of MINs

Using the previously defined basic net $\mathsf{N}_{SE}(c, m_{max})$ a Petri net model of multistage interconnection networks can be established. This section particularly describes the Petri net semantics of an $N\times N$ Modified Data Manipulator $\mathsf{MDM}(c, n, m_{max})$, which consists of n stages of $c\times c$ switching elements.

We define, by induction, a semantical function $\mathcal{PN}$ associating high-level Petri nets to MDMs: $\mathcal{PN}(\mathsf{MDM}(c, n, m_{max}))$ is a net called $\mathsf{N}^n(c, m_{max})$.

The construction will be such that all net elements, i.e. places and transitions, will have double-indexed names (identified by their annotation): the index at the top gives the level of construction by natural (decimal) numbers from 1 to n. All elements in a column will have the same index at the top. Note that this index enumerates the columns (representing the stages) from right to left while the stage numbers increase from left to right, as depicted in Figure 1. The lower index reflects a top down enumeration of the elements in a column by numbers interpreted as words β in the c-ary alphabet[2] $\mathbb{C} = \{0, \ldots, c-1\}$. This naming is based on some particular renaming functions: they change the names of places and transition by renaming the indices.

[2] Thus, concatenating c-ary numbers η with β yields $\eta\beta$. No lower index is considered as empty word $\beta = \varepsilon$, thus concatenating η with the empty word yields η. We can also factorize β in e.g. $\alpha\beta'\alpha'$, where $\alpha, \alpha' \in \mathbb{C}$ are the first and last letter of the word β, respectively.

For a given high-level net $\mathsf{N} = (P(\mathsf{N}), T(\mathsf{N}), A, \gamma)$ we define $\mu_{\text{start}}(\mathsf{N})$, $\mu^i_\eta(\mathsf{N})$, $\nu_\eta(\mathsf{N})$, $\vartheta(\mathsf{N})$, and $\rho(\mathsf{N})$ as follows.

For all $x_\beta \in P(\mathsf{N}) \bigcup T(\mathsf{N})$ we set

$$\mu_{\text{start}}(x_\beta) = x^1_\beta$$

i.e. the top index "1" is added to the original name, for instance $\mu_{\text{start}}(t) = t^1$; also

$$\mu^i_\eta(x_\beta) = x^i_{\eta\beta}$$

Here, the top index "i" is added and η prefixed to the existing lower index, for instance $\mu^3_{10}(r_{11}) = r^3_{1011}$.

For all $x^i_\beta \in P(\mathsf{N}) \bigcup T(\mathsf{N})$ (a top index already exists) we set

$$\nu_\eta(x^i_\beta) = x^i_{\eta\beta}$$

Here, the top index is unchanged and η is prefixed to the lower one.

$$\begin{aligned} \vartheta(x^i_\beta) &= b^i_\beta \qquad \text{if } x^i_\beta = r^i_\beta \\ &= x^i_\beta \qquad \text{otherwise} \end{aligned}$$

saying that **r**ight places, r, should become middle places, b, (simple **b**uffer places).

$$\begin{aligned} \rho(x^i_\beta) &= b^{i+1}_{\alpha'\beta'\alpha} \quad \text{if } x^i_\beta = \ell^i_{\alpha\beta'\alpha'} \quad \text{and} \quad \alpha, \alpha' \in \mathbb{C} \\ &= x^i_\beta \qquad \text{otherwise} \end{aligned}$$

saying that **l**eft places, ℓ, should become middle places, b. The index is changed according to the rule of stage interconnections between Stage 0 and Stage 1 of MDMs given in Section 2 by swapping the two outer digits.

A composition of such functions is expressed as usual by $\circ$, for instance $\rho \circ \nu_1(\ell^2_{10}) = \rho(\ell^2_{110}) = b^3_{110}$ and $\vartheta \circ \mu^3_{01}(r_1) = \vartheta(r^3_{011}) = b^3_{011}$.

The Petri net model can be given now by induction according to the inductive MDM scheme presented in Section 2.

Step 1: Let us define $\mathsf{N}^1(c, m_{max})$ modeling the smallest possible MDM, which is of size $c \times c$, just by renaming the basic net with μ_{start}:

$$\mathcal{PN}(\mathsf{MDM}(c, 1, m_{max})) \;=\; \mathsf{N}^1(c, m_{max}) \;=\; \mu_{\text{start}}(\mathsf{N}_{SE}(c, m_{max}))$$

By **induction hypothesis**, it is assumed that the nets $\mathsf{N}^j(c, m_{max})$ with $1 \leq j \leq i$ already exist.

Step i+1: We construct $\mathsf{N}^{i+1}(c, m_{max})$ from two intermediate nets: a net N' representing Stage 0 can be established by c^i copies of the basic net, correctly renamed from top to bottom by μ^{i+1}_η (with $\eta < c^i$) and then by ϑ to signalize that the right side is ready to be merged:

$$\mathsf{N}' = \mathop{\Big\|}_{0 \leq \eta < c^i} \vartheta \circ \mu^{i+1}_\eta(\mathsf{N}_{SE}(c, m_{max}))$$

Further on, c copies of the net $\mathsf{N}^i(c, m_{max}))$ are needed, correctly renamed from top to bottom by ν_η (adding one digit η to obtain a numbering from 0 to c^i) and then by ρ to signal that the left side is ready to be merged. The result is called N'' representing Stages 1 to i:

$$\mathsf{N}'' = \mathop{\Big\|}_{0 \leq \eta < c} \rho \circ \nu_\eta(\mathsf{N}^i(c, m_{max}))$$

Both nets N' and N'' have places named b_η^{i+1} for $0 \leq \eta < c^i$. By taking the parallel composition of them, the places with the same name will be merged. Thus we build $\mathsf{N}^{i+1}(c, m_{max})$ by setting:

$$\mathcal{PN}(\mathsf{MDM}(c, i+1, m_{max})) \quad = \quad \mathsf{N}^{i+1}(c, m_{max}) \quad = \quad \mathsf{N}' \parallel \mathsf{N}''$$

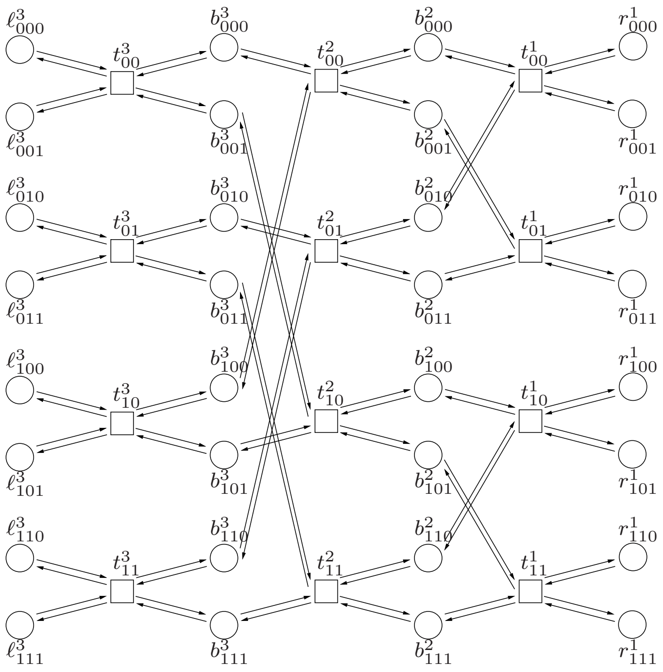


Fig. 6. Petri net model of an $\mathsf{MDM}(2, 3, m_{max})$

Figure 6 depicts an example, where the Petri net semantics of an 8×8 MDM consisting of 2×2 SEs is shown. The inscriptions of the arcs are omitted to keep the figure readable (they are the original ones, because arcs are never renamed in the construction).

The Petri net semantics of any $N\times N$ MDM can now be established by applying the given definition with parameter n such that $c^n = N$.

5 Simulation

Network topologies like MINs are mainly examined concerning their behavior and performance. Simulating the presented model deals with this task.

5.1 Incorporated Packet Generators and Packet Flushes

To produce test data for the simulation of the net model $\mathsf{N}^n(c, m_{max})$ with $c^n = N$, a packet generator model has to be added. As test data should arrive at all N left places of this net, we will consider N copies of the packet generator basic net $\mathsf{PG}(c, m_{max})$, and rename them for connection to these places. Now, the size of the required packets is known, so we consider their places ℓ of type $BList_n(c, m_{max})$ (with $c^n = N$) and all copies having uniformly the same guard γ. We can use one of the renaming functions, μ_η^n , already introduced in the previous section. Let us call $\mathsf{N}_{PG}^n(c, m_{max})$ the entire net we have to add:

$$\mathsf{N}_{PG}^n(c, m_{max}) = \mathop{\Big\|}_{0 \leq \eta < c^n} \mu_\eta^n(\mathsf{PG}(c, m_{max}))$$

In the same manner we have to add flushes at the right (destination) side of the model to removed the received data. For this, we take N copies of the basic net $\mathsf{PF}(c, m_{max})$, whose top indices are already the good ones, having places r^1 and transitions t^0. Then we enumerate them from top to bottom by using the adequate renaming functions ν:

$$\mathsf{N}_{PF}^n(c, m_{max}) = \mathop{\Big\|}_{0 \leq \eta < c^n} \nu_\eta(\mathsf{PF}(c, m_{max}))$$

Putting the three nets side by side and merging the corresponding places ℓ_η^n and r_η^1, respectively, yields the desired connection. We will add an initial marking, by putting $\bullet$ in all entry places and ε in all other places, i.e., in all buffer places. Thus

$$\begin{aligned} M_0(p) &= \bullet \qquad && \text{if } p = p_{e_\eta^n} \text{ with } 0 \leq \eta < c^n \\ &= \varepsilon && \text{otherwise} \end{aligned}$$

The obtained net system

$$\Sigma^n(c, m_{max}) = (\mathsf{N}_{PG}^n(c, m_{max}) \| \mathsf{N}^n(c, m_{max}) \| \mathsf{N}_{PF}^n(c, m_{max}), M_0)$$

will be executed, analyzed and evaluated in the sequel.

5.2 Execution

An investigation of the network behavior is usually started with an empty network, well represented by the chosen initial marking by putting ε in all buffer places.

Let us first establish some useful properties.

Proposition 4: All reachable markings of $\Sigma^n(c, m_{max})$ satisfy that each place contains exactly one token.

Proof: Clearly true, given the initial marking where each place contains exactly one token and the definition of all transitions in the net which are all connected with one input and one output arc to each one of their surrounding places.

Proposition 5: At each reachable marking of $\Sigma^n(c, m_{max})$, each transition in the net is firable.

Proof: Immediate consequence from Propositions 1, 2, 3 and 4.

Proposition 5 does not establish a bad property in the way that it allows all possible interleavings of transitions. Such an interleaving semantics will never be considered here, because it would not correspond to anything in the modeled MIN. We have to find and are able to find a special execution mode of the Petri net which corresponds naturally to a network clock cycle under the principle of *store-and-forward switching*, cf Section 2.

At each moment in the evolution of this net system all sets of transitions that do not share surrounding places are concurrently firable. This allows us to group certain sets of transitions in steps, which can be fired at the same time.

In fact, in our net system such a clock cycle of a n-stage MIN can be naturally simulated by $n + 2$ steps using step semantics.

Due to the global backpressure algorithm, packet switching is performed in MINs stage by stage, starting with the last one, to determine the availability of succeeding buffers. But nevertheless, all switches of the same stage operate in parallel. Similarly, we will group transitions of the same column, i.e. of each stage i to *simple steps*

$u^i = \{t^i_\beta | 0 \leq \beta < N\}$ with $0 \leq i \leq n$,

as well as those transitions of the packet generation column to a step

$u_{pg} = \{t'_\beta | 0 \leq \beta < N\}$.

Let us remark, that other ways to group transitions in steps are possible from a Petri net point of view, like the step of all transitions of odd stages and the step of all transitions of even stages. But once more, their executions would not correspond at all to MINs, and would in particular violate their switching principle. Thus, such steps will never be considered.

Now, let us consider first a step sequence defined by the firing of the previously defined simple steps from right to left starting with the initial marking. There exist some appropriate bindings and intermediate markings such that

$$M_0((u^0 : \sigma^0) > M_0^0((u^1 : \sigma^1) > M_0^1((u^2 : \sigma^2) > M_0^2(\quad \ldots \quad > \\ M_0^{n-1}((u^n : \sigma^n) > M_0^n((u_{pg} : \sigma') > M_1 \, .$$

The first $n+1$ steps are ordered from 0 to n such that in step i, all transitions t^i of Stage $n - i$ (or column i) fire together (Stage n virtually represents the receivers following the network outputs). These steps realize the simulation of the *global backpressure* mechanism of a MIN, which is usually sequentialized by first moving the packets of the input buffers of the succeeding network stage before it is decided at the current stage whether a packet can be moved towards them.

Step 0, the starting one, puts ε in the most right places. In step $n + 1$, the network load is modeled by firing u_{pg}. Let us recall that we have chosen in Section 3.4 a guard for transitions t'_β which adds a (possibly empty) list of tags of arbitrary size as long as the maximum length is not exceeded. This simulates a random network load. The transition guard γ can (and should) be changed to one of the other choices if we need to model any other desired network load.

We consider the entire step sequence as a *global step* leading from M_0 to M_1 corresponding to an entire clock cycle. Then, this global step can be iterated as often as necessary (for instance till any desired confidence level is reached) to obtain arbitrarily long simulations:

$$M_i((u^0 : \sigma_i^0)(u^1 : \sigma_i^1)(u^2 : \sigma_i^2) \ldots (u^n : \sigma_i^n)(u_{pg} : \sigma_i') > M_{i+1}$$

with $i > 0$. Thus we can refer to marking M_i^k as marking that is reached after the firing of u^k in the i-th global step.

By consequence, this generalized step semantics reveals to be the adequate execution mode of the model, according to the global backpressure mechanism.

5.3 Measures

The aim of modeling networks like MINs is usually to determine the performance of these networks. Performance measures like mean throughput, mean buffer queue lengths, or mean delay time are of interest. In the literature, this is estimated by stochastic analytical methods or by simulation, cf. [18,20]. Up to now, both methods (simulation and much more the analytical methods) suffered from the very complex task to establish the model: for instance in case of applying Markov chains, a large system of equations must be set up. This large system of equations, which is usually represented by matrices, must be handled by a computer to be solved. Due to the fast growing matrices if network size is increased and no model simplification is performed, only very small networks ($N \ll 10$) can be solved and performance measures obtained if using stochastic analytical methods.

We will present here, how these measures can be obtained in our context of modeling, just by establishing the model by induction and by tracing a sufficient amount of observations under the proposed step semantics.

Remember that in each buffer place p, $M(p)$ is always a unique token of type list of binary numbers (by Proposition 3). In the sequel, we denote by Len the function giving the length of such a list of numbers. Furthermore, the number of symbols “1” in the entire list $M(p)$, is as usual denoted by $|M(p)|_1$.

Because the model deals with stochastic events (e.g. conflicts between packets at SE inputs are randomly solved), several observations must be performed to achieve a certain accuracy in the results. The proposed step semantics is repeated till any predefined confidence level is reached.

All markings quoted below refer to executions under the global backpressure mechanism as described in the last subsection.

- The **mean queue length** of a particular buffer is represented by the average length of the token held by the corresponding place. Again: a sufficient number of simulation results must be considered to reach a reasonable confidence level. We could observe one place during a sufficiently long execution or even better, all places in one column k (Stage $n - k$) together in case of symmetric traffic. The average value of $\sum_\eta (Len(M_i(b^k_\eta)))$ (called $Len^k(M_i)$) for i just needs to be calculated and added to the confidence statistics.

- The **mean throughput** denotes the average number of packets that leave (or enter) a network per clock cycle and per network output (or input, respectively). Thus, the presented model determines the mean throughput of the modeled MIN by summing up (over all elements in a step) the number of markings changed in the observed places. Thus, we control how many list entries are deleted indicating that a packet is moved.

  Precisely, for one (the i-th) global step, $S_{input}(i) = \frac{1}{N} \cdot \sum_\eta [Len(M_{i+1}(\ell^n_\eta)) - Len(M^n_i(\ell^n_\eta))]$ yields the **throughput at the inputs** for this step. A sufficient number of observations determines the average throughput at the inputs considering a reasonable confidence level.

  The list in each right place r^1_η is either empty or of length $c^0 = 1$, i.e. consist of a single symbol 1 representing a packet that leaves the modeled MIN (before being flushed in the next global step). To determine the throughput at the outputs, these lists (or 1s) must be observed: $S_{output}(i) = \frac{1}{N} \cdot \sum_\eta Len(M_i(r^1_\eta))$ yields the **throughout at the outputs** for the i-th step. The average throughput at the outputs with a reasonable confidence level results by collecting a sufficient number of observations.

  Often, the **worst case scenario** of a congested network is of particular interest. Such a network state is achieved by permanently offering packets to the network inputs. We could study this by choosing an appropriate guard γ' for the PG transition as defined in the options (1) or (4), cf. subsection 3.4.
- The **mean delay time** denotes the average time a packet spends in the MIN from entering it to leaving it. Taking advantage of Little's Law, the delay time at the i-th global step can be calculated by the ratio of the number of packets in the MIN to the throughput at all outputs. But because multicasts occur, the number of packets in the MINs actually represents the number of related unicast packets. That means a multicast packet that is destined to g network outputs must be counted as g unicast packets. The number of such unicast packets is given by the number of "1" in the headers of all packets that are in the MIN. Little's Law results in a delay for the i-th step of $d(i) = (\sum_\eta [|M_i(\ell^n_\eta)|_1 + \sum_k |M_i(b^k_\eta)|_1])/(N \cdot S_{output}(i))$. A sufficient number of observations yields the mean delay time considering a given confidence level.

The given equations defining the calculus of several measures from observations are not necessarily implemented as they are given above. Faster calculations can be obtained by more sophisticated realizations. For instance, checking all places for their list entries to determine the delay time is not very efficient. An alternative approach would be to count the number of destinations of newly generated packets at each clock cycle directly by the net. A place may act as global counter of this number and subtract the number of received packets at the network outputs in the same cycle. Then, the counter holds the current number of unicast-equivalent packets in the MIN needed to determine the delay.

6 Conclusion and Ongoing Work

Systems on chip and multicore processors emerged for the last years. The required networks on chips can be realized by multistage interconnection networks. This paper presented a formal semantics of MINs to simulate and evaluate them. Particularly, multicast traffic patterns could be handled. Multicast traffic is an important traffic pattern if NoCs for multicore processors are investigated. In such networks, cache coherence strategies need multicast features for instance.

The semantics was inspired by the high-level version of the Petri box algebra which allows to represent concurrent communication systems in a totally compositional way. A dedicated small net class was formed. It led to a basic net used as a starting point to model MINs. The models were established by induction. Renaming the basic net built the model of the smallest MIN, which consists of a single stage. Models of larger MINs resulted just from a composition of basic nets as well as of MINs of one stage less.

Combining the entire model of the MIN in question with models of packet generators to feed the network with packets and packet flushes to remove received packets at the network outputs has risen a Petri net model that helps to investigate the behavior of MINs. Step semantics was very naturally applied to simulate the network behavior.

Our presented model provides a basis for performance evaluation and comparison of various MIN architectures that deal with multicast traffic patterns. If global backpressure mechanism is applied, a network cycle is modeled by $n + 2$ steps, each one consisting of the concurrent firings of the related transitions t^i in one step. That means the presented model allows to model a clock cycle by only $n + 2$ steps: a simulation complexity of $\mathcal{O}(n) = \mathcal{O}(\log_c N)$ results. The previous stochastic Petri net models of [15] suffered from a complexity of $\mathcal{O}(N \log_c N)$. The originality of our approach is to use a compositional net class, and so we are able to propose a generic modeling scheme, which applies for each MIN size and each crossbar size.

Recently we incorporated the new semantics into the toolkit *Snakes*, developed in our research group at Université Paris 12, establishing a prototype of our MIN model. The construction of a MIN is very fast for small n, as expected, and takes about 3 hours for a MDM(3,7,10) with $N = 2187$ or 5 hours for a MDM(2,11,10) with $N = 2048$. For those big models we can observe in one hour about one thousand global steps in the described semantics. We are now able to compute the mean throughputs, mean queue length, and mean delay times. Corresponding algorithms to determine the confidence level of results also need to be incorporated in the prototype, for instance the Akaroa tool [8]. Then, the tool will allow the evaluation of the measures in question.

Another advantage arises from the way how packet generators are modeled. Concerning the packet destinations, any traffic pattern can be implemented (by just redefining the guard of the PG net) representing for instance hot spot traffic. Some non-standard packet generation to simulate for instance a special kind of congested network where the same address is permanently produced, could also be expressed by an appropriate guard and allows to study worst case throughput.

References

1. Alderighi, M., Casini, F., D'Angelo, S., Salvi, D., Sechi, G.R.: A fault-tolerant FPGA-based multi-stage interconnection network for space applications. In: Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA'02), pp. 302–306 (2002)
2. Bashirov, R., Crespi, V.: Analyzing permutation capability of multistage interconnection networks with colored Petri nets. In: Information Sciences, vol. 176(21), pp. 3143–3165. Elsevier, Amsterdam (2006)
3. Best, E., Devillers, R., Koutny, M.: Petri Net Algebra. EATCS Monographs on TCS, Springer, Heidelberg (2001) ISBN 3-540-67398-9
4. Benini, L., De Micheli, G.: Networks on chips: A new SoC paradigm. IEEE Computer 35(1), 70–80 (2002)
5. Best, E., Frączak, 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 Informatica, vol. 35, Springer,Heidelberg (1998)
6. Bertozzi, D., Jalabert, A., Murali, S., Tamhankar, R., Stergiou, S., Benini, L., De Micheli, G.: NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. IEEE Transactions on Parallel and Distributed Systems 16(2), 113–129 (2005)
7. Dally, W.J., Towles, B.: Route packets, not wires: On-chip interconnection networks. In: Proceedings of Design Automation Conference (DAC 2001), pp. 684–689 (2001)
8. Ewing, G., Pawlikowski, K., McNickle, D.: Akaroa2: Exploiting network computing by distributing stochastic simulation. In: Proceedings of the European Simulation Multiconference (ESM'99), pp. 175–181 (1999)
9. Guerrier, P., Grenier, A.: A generic architecture for on-chip packet-switched interconnections. In: Proceedings of IEEE Design Automation and Test in Europe (DATE 2000), pp. 250–256. IEEE Press, New York (2000)
10. Jantsch, A.: Models of computation for networks on chip. In: Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD'06), joint invited paper ACSD and ICATPN, pp. 165–176 (2006)
11. Klaudel, H., Pommereau, F.: Asynchronous links in the PBC and M-nets. In: ASIAN 1999. LNCS, vol. 1742, Springer, Heidelberg (1999)
12. Kruskal, C.P., Snir, M.: A unified theory of interconnection network structure. Theoretical Computer Science 48(1), 75–94 (1986)
13. Pelz, E., Tutsch, D.: Modeling multistage interconnection networks of arbitrary crossbar size with compositional high level Petri nets. In: Proceedings of the 2005 European Simulation and Modelling Conference (ESM 2005),Eurosim, pp. 537–543. (2005)
14. Turner, J., Melen, R.: Multirate Clos networks. IEEE Communications Magazine 41(10), 38–44 (2003)
15. Tutsch, D., Hommel, G.: Performance of buffered multistage interconnection networks in case of packet multicasting. In: Proceedings of the 1997 Conference on Advances in Parallel and Distributed Computing (APDC'97), Shanghai, pp. 50–57. IEEE Computer Society Press, Washington (March 1997)
16. Tutsch, D., Hommel, G.: High performance low cost multicore No C architectures for embedded systems. In: Proceedings of the International Workshop on Embedded Systems – Modeling, Technology and Applications, pp. 53–62. Springer, Heidelberg (June 2006)
17. Tutsch, D., Lüdtke, D.: Compatibility of multicast and spatial traffic distribution for modeling multicore networks. In: Proceedings of the 13th International Conference on Analytical and Stochastic Modelling Techniques and Applications (ASMTA 2006); Bonn, pp. 29–36. IEEE/SCS (2006)

18. Tutsch, D., Lüdtke, D., Kühm, M.: Investigating dynamic reconfiguration of network architectures with CINSim. In: Proceedings of the 13th Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems 2006 (MMB 2006); Nürnberg, pp. 445–448, VDE (March 2006)
19. Wolf, T., Turner, J.: Design issues for high performance active routers. IEEE Journal on Selected Areas of Communications 19(3), 404–409 (March 2001)
20. Yang, Y.: An analytical model for the performance of buffered multicast banyan networks. Computer Communications 22, 598–607 (1999)
21. Yang, Y., Wang, J.: A class of multistage conference switching networks for group communication. IEEE Transactions on Parallel and Distributed Systems 15(3), 228–243 (2004)

Name Creation vs. Replication in Petri Net Systems*

Fernando Rosa-Velardo and David de Frutos-Escrig

Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
{fernandorosa,defrutos}@sip.ucm.es

Abstract. We study the relationship between name creation and replication in a setting of infinite-state communicating automata. By name creation we mean the capacity of dynamically producing pure names, with no relation between them other than equality or inequality. By replication we understand the ability of systems of creating new parallel identical threads, that can synchronize with each other. We have developed our study in the framework of Petri nets, by considering several extensions of P/T nets. In particular, we prove that in this setting name creation and replication are equivalent, but only when a garbage collection mechanism is added for idle threads. However, when simultaneously considering both extensions the obtained model is, a bit surprisingly, Turing complete and therefore, more expressive than when considered separately.

1 Introduction

Extensive work has been developed in the last years in the field of multithreaded programs [21]. In general, once thread synchronization is allowed, reachability becomes undecidable [18], so that the effort is devoted to compute overapproximations of the set of reachable markings. For instance, [2] studies the case where threads are finite state, [3] considers a framework to statically analyze multithreaded systems with a fixed number of infinite-state communicating threads, and in [4] that work is extended to cope with dynamic creation of threads.

Dynamic name generation has also been thoroughly studied, mainly in the field of security and mobility [13]. Paradigmatic examples of nominal calculi are the π-calculus [16], and the Ambient Calculus [6]. Since the early versions of all these calculi are Turing complete, numerous efforts have been focused on the static analysis of undecidable properties [7, 17] and in the restriction of the original formalisms [5, 27] to get more manageable languages where some interesting properties become decidable.

In this paper we investigate the relationships between name creation and replication in this setting of infinite-state communicating automata, that in our

* Work partially supported by the Spanish projects DESAFIOS TIN2006-15660-C02-02, WEST TIN2006-15578-C02-01 and PROMESAS-CAM S-0505/TIC/0407.

case are Petri nets. By name creation we mean a mechanism to generate fresh names that can be communicated between components, with no relation between them other than equality or inequality, which have been called pure names in [13]. By replication we understand the capability of processes of creating an exact replica of themselves, though always starting its execution in a fixed initial state. Once created, these replicated processes evolve in an independent way, though possibly interacting with other processes.

We will prove that these two mechanism are equivalent, in the sense that they simulate each other. On the one hand, names can be used to represent the states of the different threads of a process that are running in parallel over a single copy of the (virtually) replicated process. On the other hand, different copies of the same component can be used to mimic the effect of adding pure names. In the proof of this equivalence it is essential to consider a garbage collection mechanism that removes idle threads, given that the corresponding garbage collection mechanism for names is implicit in the model (the set of used names is not an explicit part of the state). In fact, reachability is undecidable for the extension with name creation (equivalently with replication and garbage collection), but decidable if no such mechanism is considered.

However, though name creation and replication are equivalent, it turns out that they do not overlap, in the sense that a model dealing simultaneously with both surpasses the strength of any of these two features, reaching indeed Turing completeness. The intuitive explanation of why this is true is that once we have names, they can be used to distinguish between what at first were indistinguishable components, so that now threads can have a unique personality.

A formalism that encompasses both dynamic name generation and replication is TDL [9]. In TDL there are primitives both for name creation and thread creation. However, no garbage collection mechanism is considered, neither for names nor for threads. As a consequence, and quite surprisingly, reachability is decidable, though coverability is undecidable.

The remainder of the paper is structured as follows. Section 2 presents the basic model, that will be the starting point of our work. Section 3 extends the basic model with a mechanism for name creation and name communication, and gives a brief insight of the expressive power of the obtained model. Section 4 extends again the basic model, but now with a replication primitive, and proves the equivalence between the two extensions. In Section 5 we consider the model obtained when introducing the two features at the same time, and prove its Turing-completeness. Section 6 presents some results regarding boundedness. Finally, Section 7 presents our conclusions and some directions for further work.

2 The Basic Model

In order to concentrate on the two features of interest we will first consider a very basic model, based on Petri Nets, which could be considered not too useful in practice, but which will be an adequate starting point for later extensions. First, let us introduce some notations that we will use throughout the paper.

Given an arbitrary set A we will denote by $\mathcal{MS}(A)$ the set of finite multisets of A, that is, the set of mappings $m : A \rightarrow \mathbb{N}$. We denote by $S(m)$ the support of m, that is, the set $\{a \in A \mid m(a) > 0\}$, $|m| = \sum\limits_{a \in S(m)} m(a)$, and by $+$ and $-$ the sum and difference operators for multisets, to distinguish them from $\cup$ and $\setminus$, the corresponding operators over sets. If $f : A \rightarrow B$ is an injective mapping and $m \in \mathcal{MS}(A)$, then we define $f(m) \in \mathcal{MS}(B)$ by $f(m)(b) = m(a)$ if $b = f(a)$ or $f(m)(b) = 0$, otherwise. We will consider a set $\mathcal{S}$ of service names, endowed with a function $arity : \mathcal{S} \rightarrow \{n \in \mathbb{N} \mid n \geq 2\}$ and we take the set of synchronizing labels $Sync = \{s(i) \mid s \in \mathcal{S},\ 1 \leq i \leq arity(s)\}$. If $arity(s) = 2$ then we will write $s?$ and $s!$ instead of $s(1)$ and $s(2)$, respectively, that can be interpreted as the offer and request of an ordinary service. We also use a set $\mathcal{A}$ of labels for autonomous transitions.

Definition 1. *A component net is a labelled Petri net $N = (P, T, F, \lambda)$ where:*

- *P and T are disjoint finite sets of places and transitions, respectively,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs of the net, and*
- *λ is a function from T to the set of labels $\mathcal{A} \cup Sync$.*

A marking M of N is a finite multiset of places of N, that is, $M \in \mathcal{MS}(P)$.

As usual, we denote by $t^\bullet$ and ${}^\bullet t$ the set of postconditions and preconditions of t, respectively, that is, $t^\bullet = \{p \mid (t,p) \in F\}$ and ${}^\bullet t = \{p \mid (p,t) \in F\}$.

Definition 2. *A basic net system is a set $\mathcal{N}$ of pairwise disjoint component nets. A marking of $\mathcal{N}$ is a set of markings of its components, one marking per component.*

For a net system $\mathcal{N}$ we will denote by P, T, F and λ the union of the corresponding elements in each net component, and we will denote simply by F the characteristic function of the set F. Analogously, we will consider markings as multisets of P. Now let us define the firing rules of the two kind of transitions.

Definition 3. *Let $\mathcal{N}$ be a basic net system and M a marking of $\mathcal{N}$. An autonomous transition, $t \in T$ with $\lambda(t) \in \mathcal{A}$, is enabled at marking M if ${}^\bullet t \subseteq M$. The reached state of $\mathcal{N}$ after the firing of t is $M' = M - {}^\bullet t + t^\bullet$.*

Therefore, autonomous transitions are exactly as ordinary transitions in P/T nets (see Fig. 1 left). This is not the case for synchronizing transitions.

Definition 4. *Let $\mathcal{N}$ be a basic net system and $s \in \mathcal{S}$ with $arity(s) = n$. The transitions in a tuple $\overline{t} = (t_1, \ldots, t_n)$ are said to be compatible for s if $\lambda(t_i) = s(i)$ for all $i \in \{1, \ldots, n\}$. We write ${}^\bullet\overline{t} = \sum\limits_{i=1}^{n} {}^\bullet t_i$ and $\overline{t}^\bullet = \sum\limits_{i=1}^{n} t_i^\bullet$.*

Definition 5. *Let $\mathcal{N}$ be a basic net system and M a marking of $\mathcal{N}$. A tuple of synchronizing transitions $\overline{t}$ is enabled at marking M if ${}^\bullet\overline{t} \subseteq M$. The reached state of $\mathcal{N}$ after the firing of $\overline{t}$ is $M' = M - {}^\bullet\overline{t} + \overline{t}^\bullet$.*

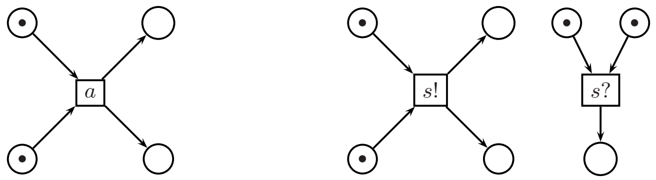


Fig. 1. Autonomous (left) and synchronizing (right) transitions

Thus, any synchronizing transition (Fig. 1 right) needs the presence of compatible enabled transitions, in which case they can be fired simultaneously. Notice that since the different nets of a basic net system do not have a name they must be addressed by communications in an anonymous way. Indeed, a net component is willing to synchronize with any nets that have enabled compatible transitions. Intuitively, it is willing to receive a service from anyone offering it. Therefore, there is no way in the present model to discriminate between different components. In fact, a process willing to synchronize can do it with any compatible counterpart, and its state after the execution will be the same whichever was that counterpart. In [12] we proved the analogous to the following result, but with a model with only two-way synchronizations.

Proposition 1. *Every Basic Net System can be simulated by a P/T net.*

The proof basically consists on having a different transition for every tuple of compatible synchronizing transitions, so that the firing of the tuple $\overline{t}$ is simulated by the firing of transition $\overline{t}$. In fact, there we proved this result for a model which also considered localities, so that each component is localized and can only synchronize with co-located components, thus proving that these locations do not introduce any further expressive power.

In the result above and some other times along the paper we assert that a model $\mathbf{M}'$ simulates another model $\mathbf{M}$. By that we mean here that for every system $\mathcal{N}$ in $\mathbf{M}$ there is $\mathcal{N}' = F(\mathcal{N})$ in $\mathbf{M}'$, where F is a computable function, such that the transition systems generated by the semantics of $\mathcal{N}$ and $\mathcal{N}'$ are isomorphic. Therefore, reachability in $\mathcal{N}$ and $\mathcal{N}'$ are equivalent. Moreover, that isomorphism also preserves the considered orders between markings, so that coverability and boundedness are also preserved. As a consequence, whenever reachability, coverability or boundedness are decidable in $\mathbf{M}'$ they are also decidable in $\mathbf{M}$.

Since reachability, coverability and boundedness are decidable for P/T nets, we have the following:

Corollary 1. *Reachability, coverability and boundedness are decidable for Basic Net Systems.*

3 Name Creation

In this section we extend the previous model with the capability of name management. Names can be created, communicated between components and used

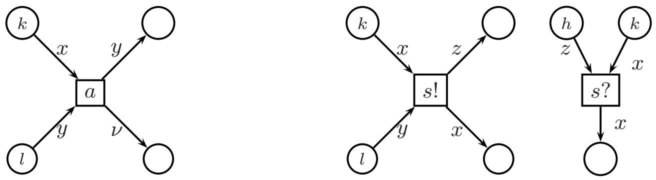


Fig. 2. Autonomous (left) and synchronizing (right) transitions

to restrict synchronizations to happen only between components that know a particular name, as we have illustrated with several examples in [24]. We can use this mechanism to deal with authentication issues in our systems. We formalize the latter by replacing ordinary tokens by distinguishable tokens, thus adding a touch of colour to our nets. In order to handle these colours, we need matching variables labelling the arcs of the nets, taken from a set Var. Moreover, we add a primitive capable of creating fresh names, formalized by means of a special variable $\nu \in Var$.

Definition 6. *A ν-net component is a labelled coloured Petri Net $N=(P,T,F,\lambda)$, where P and T are finite disjoint sets of places and transitions of the net, respectively, $F:(P\times T)\cup(T\times P)\to Var$ is a partial function, $\lambda : T \to \mathcal{A}\cup Sync$ is a function labelling transitions, such that for every $t\in T$:*

1. *$\nu \notin pre(t)$,*
2. *If $\lambda(t)\in\mathcal{A}$ then $post(t)\setminus\{\nu\}\subseteq pre(t)$,*
3. *If $\lambda(t)\in Sync$ then $\nu\notin Var(t)$,*

where ${}^\bullet t=\{p\mid (p,t)\in Dom(F)\}$, $t^\bullet=\{p\mid (t,p)\in Dom(F)\}$, $pre(t)=\{F(p,t)\mid p\in {}^\bullet t\}$, $post(t)=\{F(t,p)\mid p\in t^\bullet\}$ and $Var(t)=pre(t)\cup post(t)$.

A ν-net component is a special kind of labelled coloured Petri Net with only one colour type for identifiers, taken from an infinite set Id, except for the special variable ν. Unlike for ordinary Coloured Petri Nets, where arbitrary expressions over some syntax can label arcs, we only allow variables, that are used to specify the flow of tokens from preconditions to postconditions. In particular, this means that only equality of identifiers can be checked by matching.[1] If t is an autonomous transition then it must be the case that every variable annotating a postcondition arc, except ν, also annotates some precondition arc. Then, the only way transitions can produce new names is by means of the variable ν attached to one of its outgoing arcs, which always produces a new name, not present in the current marking of the system.

Definition 7. *A marking of a ν-net $N=(P,T,F,\lambda)$ is a function $M:P\to\mathcal{MS}(Id)$. We denote by $S(M)$ the set of names in M, that is, $S(M)=\bigcup\limits_{p\in P}S(M(p))$.*

Definition 8. *A ν-net system $\mathcal{N}$ is a set of disjoint ν-net components. A marking of $\mathcal{N}$ is a collection of markings of its components, one marking each.*

[1] In fact, analogous results could be obtained if we could also check inequality.

In fact, ν-net systems with a single component correspond to the minimal OO-nets defined in [14]. As for any kind of coloured Petri nets, transitions are fired relative to a mode $\sigma : Var(t) \to Id$, that chooses among the tokens that lie in the precondition places. We will denote modes by $\sigma, \sigma', \sigma_1, \sigma_2, \ldots$

Definition 9. *Let $\mathcal{N}$ be a ν-net system, $t \in T$ with $\lambda(t) \in \mathcal{A}$ and M a marking of $\mathcal{N}$. A transition t is enabled with mode σ if $\sigma(\nu) \notin S(M)$ and for all $p \in {}^\bullet t$, $\sigma(F(p,t)) \in M(p)$. The reached state of $\mathcal{N}$ after the firing of t with mode σ is the marking M' given by $M'(p) = M(p) - \{\sigma(F(p,t))\} + \{\sigma(F(t,p))\} \;\; \forall p \in P$.*

Autonomous transitions work mainly as the ordinary transitions in coloured nets (see Fig. 2 left). The only novelty is the presence of the variable ν, which is specially treated in the firing rule: The condition $\sigma(\nu) \notin S(M)$ causes the creation of fresh (equal) identifiers in all the places reached by arcs labelled by that special variable.

For a tuple of synchronizing transitions $\overline{t} = (t_1, \ldots, t_n)$ we denote by $post(\overline{t}) = \bigcup_{i=1}^{n} post(t_i)$, $pre(\overline{t}) = \bigcup_{i=1}^{n} pre(t_i)$ and $Var(\overline{t}) = post(\overline{t}) \cup pre(\overline{t})$.

Definition 10. *Let $\overline{t} = (t_1, \ldots, t_n)$ be a tuple of synchronizing transitions of a ν-net system. We say the transitions in $\overline{t}$ are compatible if:*

1. *$\lambda(t_i) = s(i)$ for some $s \in \mathcal{S}$ with $arity(s) = n$,*
2. *$post(\overline{t}) \setminus \{\nu\} \subseteq pre(\overline{t})$*

The compatibility conditions are still merely syntactical: All the transitions in the tuple must meet together the same constraint imposed to autonomous transitions (see Fig. 2 right). A mode for $\overline{t}$ is a map $\sigma : Var(\overline{t}) \to Id$.

Definition 11. *Let $\mathcal{N}$ be a ν-net system and M a marking of $\mathcal{N}$. We say that the tuple of compatible transitions $\overline{t} = (t_1, \ldots, t_n)$ is enabled with mode σ if for all $p \in {}^\bullet\overline{t}$, $\sum_{i=1}^{n} \{\sigma(F(p,t_i))\} \subseteq M(p)$. The reached state of $\mathcal{N}$ after the firing of $\overline{t}$ with mode σ is the marking M', given by*

$$M'(p) = M(p) - \sum_{i=1}^{n} \{\sigma(F(p,t_i))\} + \sum_{i=1}^{n} \{\sigma(F(t_i,p))\} \;\; \forall p \in P$$

By means of synchronization we achieve both name communication and restriction of communications by name matching. If $\overline{t} = (t_1, t_2)$, the former is obtained by using a variable in $post(t_1) \setminus pre(t_1)$ and in $pre(t_2)$ (or vice versa), as the variable z in Fig. 2 right. The latter is obtained by using the same label both in $pre(t_1)$ and $pre(t_2)$, which forces the matching between the corresponding tokens, as the variable x in Fig. 2 right.

In [24] we proved several interesting (un)decidability results for a model we call Mobile Synchronizing Petri Nets, that are essentially these ν-net systems, but again with some syntactic sugar supporting a flat kind of mobility and allowing only two-way synchronizations. In particular, though reachability turns out to be undecidable when adding names, as proved in [14] for minimal OO-nets, coverability remains decidable, meaning that the expressive power of this model lies somewhere in between ordinary P/T nets and Turing machines. The latter was proved by taking into account the abstract nature of created names. More precisely, we proved that they are well structured systems [11] when we consider the order defined by $M_1 \sqsubseteq_\alpha M_2 \Leftrightarrow$ if there is some M' such that $M_1 \equiv_\alpha M'$ (they are equal up to renaming of identifier tokens) and $M'(p) \subseteq M_2(p)\ \ \forall p \in P$. All these results are trivially transferred to the model considered in this paper.

4 Replication

So far, components were not an essential feature of any of the models just considered. In fact, any basic net system or ν-net system could be flattened to an equivalent one with a single component. However, we have preferred to maintain components in order to have an incremental presentation, and because they are the basis of the agent based architecture of mobile systems that motivated our study. In this section we introduce a replication primitive, that creates an identical copy of the net component that executes it, marked in some fixed way. This primitive makes very desirable the structuring of the system by means of components. Replication, together with synchronization, can be used to implement a spawning primitive, that creates a different component taken from a finite set.

Definition 12. *A Replicated Net (RN) is a labelled Petri net $N = (P, T, F, \lambda)$ where:*

- *P and T are finite disjoint sets of places and transitions, respectively,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs of the net,*
- *λ is a function from T to the set of labels $\mathcal{A} \cup Sync \cup \mathcal{MS}(P)$.*

A marking M of N is a finite multiset of places of N.

The only syntactical difference with basic nets is that now some transitions may be labelled by a multiset of places, that is, by a marking. That multiset corresponds to the initial marking of the replicated net that is created when firing such a transition.

We represent the presence in the system of several copies of a net by considering several markings of that net, so that now each $N \in \mathcal{N}$ does not represent a single net component, but one of the possible types of nets that can appear in the system. Therefore, unlike for ordinary basic nets, a marking is not just a collection of individual markings, one marking per component, but a multiset of markings, and not necessarily one marking per component, but any natural.

Definition 13. *An RN system $\mathcal{N}$ is a set of pairwise disjoint RNs. A marking $\mathcal{M}$ of $\mathcal{N}$ is a finite multiset of markings of its components.*

Alternatively, we could expand the state of a replicated net system, getting an equivalent system with several copies of each net in the initial system, each marked by an ordinary marking. Then, the firing of a replication transition would

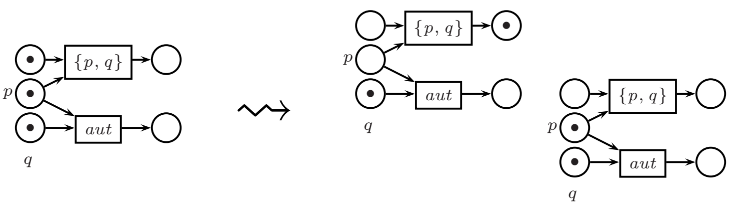


Fig. 3. RN system

add a new component to the system, thus modifying its architecture. However, in order to clearly justify our positive results about the decidability of some properties it is crucial to stress the fact that the set of types of the components of the system remains the same, as explicitly captured by our definitions.

Given an RN system $\mathcal{N}$, marked by $\mathcal{M}$, we will denote by $\mathcal{M}(N)$ the marking of the subsystem of $\mathcal{N}$ composed only of copies of N, so that in order to completely specify a marking $\mathcal{M}$ it is enough to specify each $\mathcal{M}(N)$ for every $N \in \mathcal{N}$. The definitions of enabled transition and firing of transitions are analogous to those for basic net systems. Next, we will present the definitions corresponding to autonomous transitions.

Definition 14. *Given an RN system $\mathcal{N}$ with $\mathcal{N} = \{N_1, \ldots, N_n\}$, $N_i = (P_i, T_i, F_i, \lambda_i)$, and $\mathcal{M}$ a marking of $\mathcal{N}$, $t \in T_i$ with $\lambda(t) \in \mathcal{A}$ is M-enabled if $M \in \mathcal{M}(N_i)$ and ${}^\bullet t \subseteq M$. The reached marking after the M-firing of t is $\mathcal{M}'$, where*

– $\mathcal{M}'(N_i) = \mathcal{M}(N_i) - \{M\} + \{M'\}$, *where* $M' = M - {}^\bullet t + t^\bullet$,
– $\mathcal{M}'(N_j) = \mathcal{M}(N_j)$ *for every* j *with* $i \neq j$.

In the previous definition, the marking of the component firing the transition is replaced in $\mathcal{M}(N)$ by the resulting marking of that firing, where N is the type of that component.

The definition of firing of a tuple of synchronizing transitions is analogous to that in the previous sections, but taking into account that now there may be several components of the same type and that the synchronizing transitions may or may not belong to the same components. To complete the presentation, we define the firing of replication transitions.

Definition 15. *Given an RN system $\mathcal{N}$ with $\mathcal{N} = \{N_1, \ldots, N_n\}$, $N_i = (P_i, T_i, F_i, \lambda_i)$, and $\mathcal{M}$ a marking of $\mathcal{N}$, $t \in T_i$ with $\lambda(t) \in \mathcal{MS}(P_i)$ is M-enabled if $M \in \mathcal{M}(N_i)$ and ${}^\bullet t \subseteq M$. The reached marking after the M-firing of t is $\mathcal{M}'$, where*

– $\mathcal{M}'(N_i) = \mathcal{M}(N_i) - \{M\} + \{M', \lambda(t)\}$, *where* $M' = M - {}^\bullet t + t^\bullet$,
– $\mathcal{M}'(N_j) = \mathcal{M}(N_j)$ *for every* j *with* $i \neq j$.

Therefore, the net firing the replication transition is changed as if it were an autonomous transition; besides, a new net of the same type is created, initially marked by $\lambda(t)$. Fig. 3 illustrates an RN system, initially with one component that can either fire an autonomous transition (labelled by *aut*) or create a replica that can only fire that autonomous transition, and chooses to do the latter.

So far, net components can only be created, but never removed, even though no tokens lie in them, thus making impossible the firing of any of its transitions (supposing, without loss of generality that every transition has at least one precondition). In such a case, we proved in [26] that both reachability and coverability are decidable. The former is proved by taking into account that markings must remember the number of created components, even if some of them are deadlocked. Since this assumption is not very realistic, next we introduce a garbage collection mechanism, that allows us to remove from markings those nets that do not currently have any token, which form a subset of deadlocked components, if we assume that every transition has at least one precondition. We do it simply by disregarding the empty marking as a possible marking of a net, as formalized in the next definition.

Definition 16. *Given two markings $\mathcal{M}$ and $\mathcal{M}'$ of an RN system $\mathcal{N}$ we will write $\mathcal{M} \equiv \mathcal{M}'$ if for all $N \in \mathcal{N}$, $\mathcal{M}(N) \equiv_N \mathcal{M}'(N)$, where $\equiv_N$ is the least equivalence relation on multisets of markings of N such that $M \equiv_N M + \{\emptyset\}$.*

If we identify markings up to $\equiv$, every time a net component becomes empty, we will ignore it. We will write g-RN to denote RN systems with garbage collection. Once again, in [26] we proved the decidability of coverability for g-RN systems by means of a simulation of them using ν-net components.

Proposition 2. *Every g-RN system can be simulated by a ν-net system.*

The proof of the previous result consists on simulating the behaviour of every copy of the same component within the same net, using a different identifier to simulate the ordinary black tokens in each of the copies, thus distinguishing between the different copies. In particular, garbage collection of empty components is obtained for free by means of the disappearance of the identifier that represents it. We guarantee that different components do not incorrectly interact with each other labelling every arc of the autonomous transitions with the same variable, so that they only manipulate tokens of the same type (representing one single net component). Moreover, creation of net components is mapped to creation of new tokens that reflect the initial marking of the new component.

Notice that if we removed every deadlocked component, instead of only the empty ones, this simulation would no longer be correct, unless we could remove from ν-net markings useless identifiers (those that cannot enable any transition). This fact illustrates that in fact the garbage collection for identifiers obtained for free due to their disappearance is not an optimal garbage collection mechanism, but a conservative one.

Therefore, every decidable problem for ν-net systems that is preserved by this simulation is also decidable for g-RN systems, so that coverability is decidable for the latter. Now we prove that the reciprocal is also true, so that undecidability results are also transferred from ν-net systems to g-RN systems.

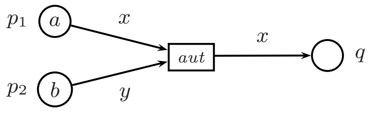


Fig. 4. ν-net without name creation

Proposition 3. *Every ν-net system can be simulated by a g-RN system.*

Proof (sketch). Given a ν-net system N, we have to simulate it using a g-RN system N^*. Without loss of generality, we assume that the ν-net system is just a ν-net without synchronizing transitions, since every ν-net system can be easily flattened to a single equivalent component.

The key idea is to use an RN component for each different name, all of the same type. This component type has the same places as N, so that a black token in a place p corresponding to the component that is simulating the identifier a stands for a token a in the place p of the original net. Fig. 5 shows the simulation of the net in Fig. 4. There, the net in the left simulates the occurrence of a in Fig. 4, and the one in the right does the same for b.

This is a straightforward way to represent the markings of N in N^*. However, the simulation of firings is quite more intricate. Given a transition of N, if every arc that is adjacent to it is labelled with the same variable, then the firing of that transition only involves one token name. If, for instance, that token is a then the RN component representing a can fire an autonomous transition that mimics that firing, moving tokens from the preconditions of t to its postconditions. However, if there is more than one variable adjacent to t then that is not possible. Consider the net in Fig. 4, with two different variables x and y next to its sole transition. It may still be the case that it is fired with x and y instantiated to the same value, so that we need an autonomous transition, labelled by $x = y$ in Fig. 5, that mimics the original transition, as in the previous case. However, t can also be fired taking two different tokens, say a and b, for x and y, respectively. In that case two net components, one representing a and one representing b, should interact by means of a pair of synchronizing transitions to simulate the firing: The one representing a should remove a token from p_1 and put it in q (action $(x \neq y)_x$ in Fig. 5), while the one representing b should just remove a token from its place p_2 (action $(x \neq y)_y$ in Fig. 5). However, since both components must be of the same type, they both must be ready to perform these three actions. That is why in the simulation shown in Fig. 5 we have three transitions, one autonomous transition and two synchronizing transitions.

Regarding name creation, we map it to component creation. Thus, every transition with outgoing arcs labelled by ν are simulated by replicating transitions. The initial marking of the new component is that with a token in every postcondition linked by a ν-arc (see Fig. 6). Notice that we have to remove in the simulation the arcs labelled by ν, since the replicating transition automatically marks those places when fired. This construction works if we assume that transitions that create names do not deal with more than two token names, which in

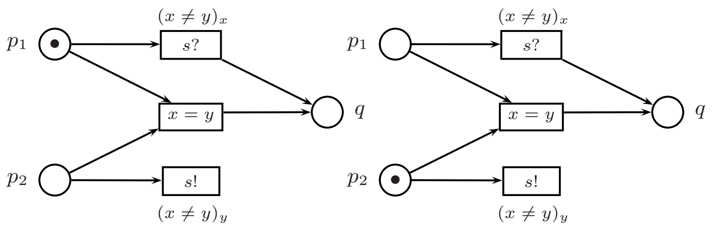


Fig. 5. Simulation of the ν-net in Figure 4 by means of g-RN systems

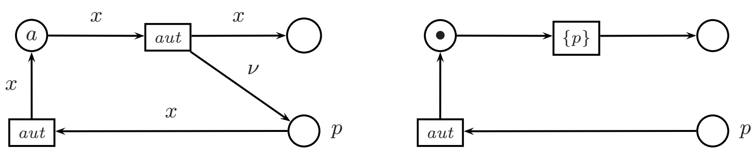


Fig. 6. ν-net with name creation and its RN simulation

fact can be done without loss of generality (otherwise, RN components should allow synchronizing transitions to create new components).

The previous construction can be generalized to transitions with an arbitrary number of variables. In the case of two variables, we have implicitly used the two partitions of the set $\{x, y\}$, namely $\{\{x, y\}\}$ and $\{\{x\}, \{y\}\}$. In general, for an arbitrary transition t, we would need to consider any possible partition of the set $Var(t)$. For each element in the partition, if it has n elements then we add n transitions, and label them with labels $s(1), \ldots, s(n)$ for some new $s \in \mathbb{S}$ with arity n, to make them compatible.

Corollary 2. *The reachability problem is undecidable and the coverability problem is decidable for g-RN systems.*

5 Name Creation + Replication

Now we consider a model in which we combine the two features in the two previous sections, that is, systems composed of nets that can both create fresh identifiers and replicate themselves. Formally, ν-RN systems are ν-net systems for which we allow an extra transition label type, so that we can label transitions with markings of components, as in the previous section, but now these markings are composed of named tokens.

Quite surprisingly, though the expressive powers obtained by adding either name creation or only replication are identical, as we have proved in the previous section, it turns out that they are somehow orthogonal, so that when combined we reach Turing-completeness, as we will prove next. Informally, we could say that both features generate bidimensional infinite state systems. In the case of ν-net components we may have an unbounded number of different tokens, each of which can appear in markings an unbounded number of times. Analogously, in the case of RN systems, we may have an unbounded number of components, each of them with a possible unbounded number of tokens. However, these two dimensional spaces are not the same, although somehow equivalent to each other, according to Prop. 2 and Prop. 3. But when we merge the three sources of infinity, only the common dimension overlaps, or following the geometric metaphor, we have two perpendicular planes, that no longer generate a bidimensional space, but the whole tridimensional space. We will see that this space is too rich, thus producing a Turing-complete formalism.

Proposition 4. *Any Turing machine can be simulated by a ν-RN System.*

Proof (sketch). Given a Turing machine, we have to simulate it using a ν-RN System. The main problem to simulate Turing machines is the representation of the potentially one-way infinite set of cells of the tape. We represent the tape by means of a doubly-linked list. Each node of the list (or cell of the tape) will be represented by a different copy of the same process, depicted in Fig. 7. This process has a memory with two kinds of data: First, it must know whether its value is 0 or 1. For that purpose, it has two places named 0 and 1, that are marked in mutual exclusion. Second, and more important, the cell must hold information about which are the previous and next cell whenever it becomes part of the tape of the machine, for what we will use identifiers. For that purpose, components that represent cells have three places (among others), *me*, *prev* and *next*, that contain three different identifiers: the name of the cell in *me*, that of the previous cell in *prev* and that of the next cell in *next*.

At any time, the tape has a final cell, that does know its own name and that of its previous cell (it has been initialized firing its *init*? transition), but not that of the next cell, which in fact does not still exist. In order to expand the tape a new cell can be created by firing the replicating transition $\{i, 0\}$. The new cell can generate its own name, after which it is willing to synchronize with its parent net (the cell at the end of the tape). Then, the synchronization of the transitions *init*! and *init*? causes the exchange of the respective names of the cells, putting them in the corresponding *next* and *prev* places, so that the net in the tape now knows the name of its next cell, and the new cell becomes the last of the tape.

Finally, the cells should also have synchronizing transitions that output the name of the previous and the next cell (transitions *left*! and *right*!, respectively), two others that output its current value (transitions $r(0)!$ and $r(1)!$) and finally four more that change it as indicated by the program of the machine (those labelled by $w(0)?$ and $w(1)?$). All these transitions are doubly-linked with the place *me*, by arrows labelled by *auth* (those labelled by $w(0)?$ and $w(1)?$ should also have those arrows, but we have omitted them in the picture for readability).

The part of the construction described so far is the same for any Turing Machine. The rest of the simulation only needs an additional net component for the control, with a place *now* containing the identifier of the cell where the head is pointing. All the synchronizations with the program mentioned in the previous paragraph are done forcing the matching between the value in the place *now* of

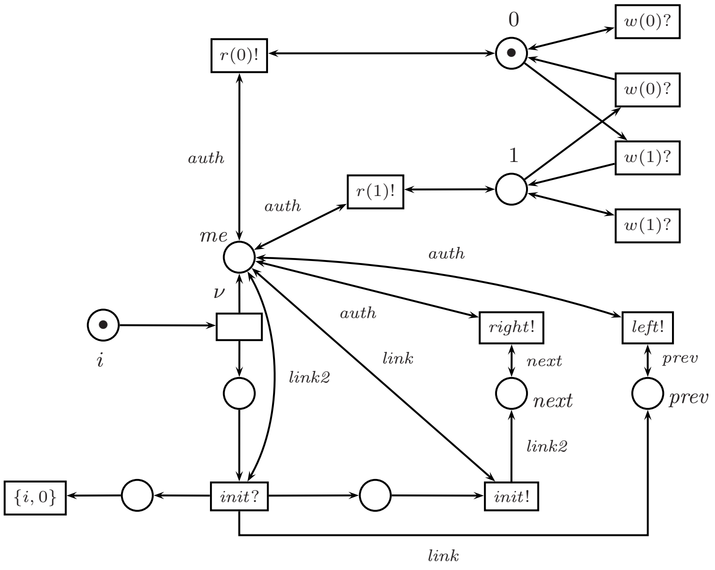


Fig. 7. Turing cell

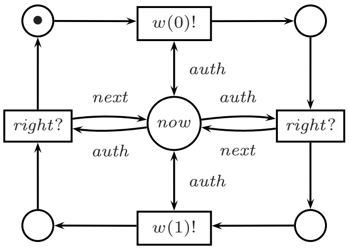


Fig. 8. Turing machine that computes 0101 . . .

the program and the value in the place *me* of the cell, by means of the variable *auth*, so that the head can only read and modify the proper cell. Of course, it should also have one place per state of the machine, marked in mutual exclusion, each of them with the corresponding actions attached. As an example, Fig. 8 shows the simulation of the head of a Turing machine that computes the infinite sequence 0101 . . .

Notice that the simulation in the proof of the previous result does not make any use of garbage collection, so that Turing completeness is achieved even without it. In fact, we are implicitly considering a kind of garbage collection for identifiers, in the sense that they could disappear from markings after the firing of some transition (as happens for instance to identifier b in Fig. 4 after firing *aut*), thus becoming reusable. We conjecture that if we avoided this automatic garbage collection for names, by including the full set of created names to the state,

$\mathcal{M}^1$	$\mathcal{M}^2$	$\mathcal{M}^3$
1	1	1
1 2	2 2	2 2
	2 3 3	3 3 3
		3 4 4 4

Fig. 9. The natural order of ν-RNs is not a wqo

we would get a situation analogous to that in [9], so that reachability is also decidable. Instead, we can prove the following undecidability result.

Corollary 3. *Coverability and reachability are undecidable for ν-RN systems.*

Proof. It is clear that reachability remains undecidable since it also was undecidable in the less general model of ν-net systems. Coverability is undecidable for ν-RN systems, or we could use the previous simulation to decide the halting problem in Turing machines, just by asking whether the marking with a token in the place representing the final state can be covered.

It is worth pointing out that though the natural orders for both ν-net systems and RN systems are well-quasi orders (wqo) [11], as a consequence of the previous undecidability result, the natural order for ν-RN systems, that extends both, cannot be wqo.

Definition 17. *Given two markings $\mathcal{M}_1$ and $\mathcal{M}_2$ of a ν-RN system $\mathcal{N}$ we define the order $\sqsubseteq$ given by $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$ if there are two injections $h_1 : \mathcal{M}_1 \to \mathcal{M}_2$ and $h_2 : S(\mathcal{M}_1) \to S(\mathcal{M}_2)$ such that for every $M \in \mathcal{M}_1$, $h_1(M)(p) \subseteq h_2(M(p))$.*

Therefore, h_1 has the role of mapping components of $\mathcal{M}_1$ to components of $\mathcal{M}_2$, while h_2 has the role of mapping identifiers in $\mathcal{M}_1$ to identifiers in $\mathcal{M}_2$. Indeed, the defined order is not a well-quasi order. To see that, it is enough to consider the simple case of systems with a single net type, with just one place, so that every marking of such a system consists on a multiset of multisets of identifiers. In this case, the previous definition is simplified to $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$ if there are two injections $h_1 : \mathcal{M}_1 \to \mathcal{M}_2$ and $h_2 : S(\mathcal{M}_1) \to S(\mathcal{M}_2)$ such that for all $M \in \mathcal{M}_1$, $h_1(M) \subseteq h_2(M)$. Let us take $Id = \mathbb{N}$ and consider the sequence of markings depicted in Fig. 9, $\mathcal{M}^i = \{M_1^i, \ldots, M_i^i, M_{i+1}^i\}$, for $i = 1, 2, \ldots$, where $M_k^i = \{k, \underset{i}{\ldots}, k\}$ for $k = 1, \ldots, i$ and $M_{i+1}^i = \{i, i+1, \underset{i}{\ldots}, i+1\}$, for which there are no indices $i < j$ such that $\mathcal{M}^i \sqsubseteq \mathcal{M}^j$. This is so because for any $i < j$, by cardinality, the only possible choice is to take h_1 and h_2 as $h_1(M_k^i) = M_k^j$ for $k = 1, \ldots, i$, and h_2 the identity, which does not work because $i \in M_{i+1}^i$ but $i \notin M_k^j$ for all $k \geq i + 1$.

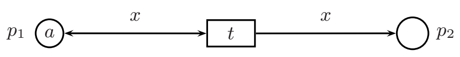


Fig. 10. Width-bounded but not depth-bounded ν-net

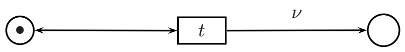


Fig. 11. Depth-bounded but not width-bounded ν-net

6 Boundedness Results

Unlike ordinary P/T nets, that have a single infinite dimension, our ν-net systems have two infinite dimensions, since identifiers can appear an unbounded number of times, and there may also be an unbounded amount of different identifiers. Therefore, we can consider three different notions of boundedness, depending on the dimensions we are considering. In this chapter we will consider ν-net systems with a single component since, as we have said before, every ν-net system can be flattened to an equivalent ν-net.

Definition 18. *Let $N = (P, T, F, M_0)$ be a ν-net.*

1. *N is bounded if there is $n \in \mathbb{N}$ such that for every reachable M and every $p \in P, |M(p)| \leq n$.*
2. *N is depth-bounded if there is $n \in \mathbb{N}$ such that for every reachable M, $a \in Id$ and $p \in P$, $M(p)(a) \leq n$.*
3. *N is width-bounded if there is $n \in \mathbb{N}$ such that for every reachable M, $|S(M)| \leq n$.*

Proposition 5. *A ν-net is bounded if and only if it is depth-bounded and width-bounded.*

As we have said before, in [24] we proved that ν-net systems[2] with order $\sqsubseteq_\alpha$ are well structured systems. Moreover, Prop. 4.5 and Lemma 4.7 prove that, in fact, they are *strictly* well structured (well structured systems with strict monotonicity). In [11] it is proved that for this kind of systems the boundedness notion induced by the considered order is decidable.

Corollary 4. *Boundedness is decidable for ν-nets.*

Therefore, according to the previous result, we can decide if a ν-net is bounded, in which case it is both depth and width-bounded according to Prop. 5. However, if it is not bounded, it could still be the case that it is width-bounded (see Fig. 10) or depth-bounded (see Fig. 11), though not both simultaneously.

Now let us see that width-boundedness is also decidable for ν-nets, for which we can reuse the proof for P/T nets.

[2] MSPN systems with abstract identifiers, as called there.

$$\begin{tikzcd}
s_1 = \{\{p_1\}\} \arrow[d, "t"] \\
s_2 = \{\{p_1, \infty(p_2)\}\} \arrow[loop below, "t"]
\end{tikzcd}$$

Fig. 12. Karp-Miller tree of the ν-net in Fig. 10

Proposition 6. *Let N be a ν-net with initial marking M_0. N is width-unbounded if and only if there are two markings M_1 and M_2 such that:*

- *M_1 is reachable from M_0 and M_2 is reachable from M_1*
- *$M_1 \sqsubseteq_\alpha M_2$*
- *$|S(M_1)| < |S(M_2)|$*

As a consequence, in order to prove that a net is width-unbounded it is enough to find two witness markings like the ones appearing in the result above. These witnesses can be found by constructing a modified version of the Karp-Miller tree that considers markings that are identified up to renaming of identifiers and some of the identifiers can appear infinitely-many times, and by cutting out branches in which the amount of different identifiers strictly grows.

Corollary 5. *The problem of deciding whether a ν-net is width-bounded is decidable.*

We have said that though the coverability problem for ν-net systems is decidable, reachability is undecidable. Of course, if the net is bounded then its space state is finite and, therefore, reachability for this class of ν-nets is decidable. In fact, this is also true for the class of width-bounded ν-nets, even though they can generate infinite state spaces. This is so because it is possible to simulate every width-bounded ν-net by means of a ν-net that does not use the ν-variable, that is, that does not create any identifier, and nets in this subclass are equivalent to P/T nets [23].

Proposition 7. *Every width-bounded ν-net can be simulated by a ν-net without name creation.*

The simulation is based on the fact that, for identifiers that can appear a bounded number of times, we can control when they are removed by explicitly having the nodes of the Karp-Miller tree of the net as places of the simulating net, marked (in mutual exclusion) whenever the ω-marking they represent cover the current marking. By knowing when an identifier is removed, they can be reused, so that at that point we can put them in a repository R of identifiers that can be used whenever a new one was created in the original net. Moreover, we only need to reuse identifiers that can appear a bounded number of times, since they are the only ones that may disappear in all possible executions (even though identifiers that appear an unbounded number of times could also eventually disappear).

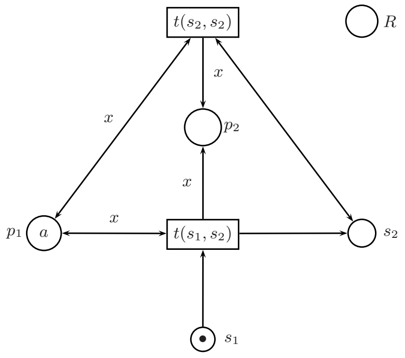


Fig. 13. ν-net without name creation that simulates the ν-net in Fig. 10

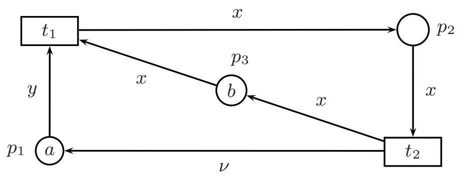


Fig. 14. Another ν-net

$$\begin{tikzcd}
s_1 = \{\{p_1\},\{p_3\}\} \arrow[d, bend right, "t_1"'] \\
s_2 = \{\{p_2\}\} \arrow[u, bend right, "t_2"']
\end{tikzcd}$$

Fig. 15. Karp-Miller tree of the ν-net in Fig. 14

As an example, let us consider the ν-net in Fig. 10 (which does not create any identifier already), whose Karp-Miller tree is depicted in Fig. 12. Since we are identifying markings up to renaming of identifiers, we can represent them as the multiset of multisets of places in which identifiers appear, one multiset of places per identifier. Then, the ν-net that results of the simulation sketched above is shown in Fig. 13. Another example is shown in Fig. 14, whose Karp-Miller tree is depicted in Fig. 15, and the simulating ν-net is that in Fig. 16. In it we use a variable ν', different from ν, that mimics the effect of ν by taking the "new" identifier from the repository.

Corollary 6. *Reachability is decidable for width-bounded ν-nets.*

Proof. Given a marking M of the ν-net N, we construct its Karp-Miller tree, which is finite, and the ν-net without name creation N^* that simulates it. Let n be a natural such that $|S(\overline{M})| \leq n$ for all $\overline{M} \in S$. If $|S(M)| > n$ then M is trivially not reachable. Let us suppose that $|S(M)| \leq n$. In that case, M is

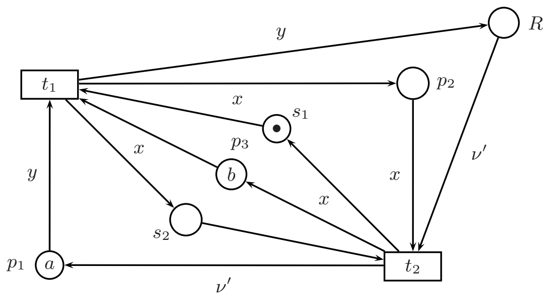


Fig. 16. ν-net without name creation simulating the ν-net in Fig. 14

reachable in N if and only if there is $\overline{M} \in S$ and a bijection $h : I \to S(M)$ such that:

– $M \sqsubseteq_\alpha \overline{M}$
– M^* is reachable in N^*, where
  • $M^*(p)(a) = M(p)(h(a))$ for all $p \in P$,
  • $M^*(\overline{M}) = 1$ and $M^*(s) = 0$ for all $s \neq \overline{M}$.

Since S is finite and there are finitely-many bijections between I and $S(M)$, reachability in N reduces to reachability in N^*, which is decidable [24].

As final remark, all the boundedness results we have seen in this section regarding identifiers can be translated to the setting with replication, thanks to Prop. 3 and Prop. 2. Therefore, the analogous version of width-boundedness, that we can call component-boundedness is decidable for g-RN systems, and reachability is also decidable for component-bounded g-RN systems.

7 Conclusions and Future Work

We have investigated the consequences of adding two different primitives to Petri Net Systems. The first one is a pure name creation primitive. The model that results from adding it was already studied in [14] and [24], where we proved that its expressive power lies in between that of ordinary Petri Nets and Turing machines. The second primitive, which was presented in [26], deals with component replication. We have proved that these two extensions can simulate each other and thus have the same power. However, when we simultaneously extend the basic model in the two directions, by incorporating both the name creation and the replication primitive, the obtained model turns out to be Turing complete.

There are several directions in which we plan to continue our research. First, we would like to know the relation between our g-RN systems (or equivalently, our ν-net systems) and some well established Petri net formalisms, whose expressive power is also in between Turing machines and Petri nets, such as Petri nets with transfer or reset arcs. We already know that both can be weakly (that is, preserving reachability) simulated using ν-nets, but we do not know if a more faithful simulation that preserves the full behaviour of the net is also possible.

Another interesting issue would be finding some restrictions to the use of the ν variable or that of replicating transitions, so that a model with constrained features but including both primitives would no longer be Turing complete, thus having similar results to those presented in Sect. 6 in the complete model. For instance, it is clear that this is the case if the number of times we can use the replication transitions is globally bounded, or equivalently, whenever we can only create a finite number of names. Certainly, the simulation of Turing machines we have presented would not work if only a bounded quantity of different names could appear at any reachable marking, even if an arbitrary number of them can be created, which suggests that coverability remains decidable in that case.

It would also be desirable to establish a complete hierarchy of models, including those appearing in [23], and other related models such as [8, 9, 15]. This hierarchy would be based on the quantity of information about the past that a configuration of a system remembers [22]. For instance, the difference between replication with and without garbage collection is that in the former case we force the configuration to remember the number of replicated componentes, which we do not do in the latter.

We are also interested in the practical implementation of the results presented in this paper. In [25] we presented a prototype of a tool for the verification of MSPN systems, which mainly corresponds to our ν-nets here. We plan to extend it to include the replication operator and coverability, boundedness and reachability algorithms, covering the cases in which these properties are decidable.

Finally, let us comment that we have found many interesting connections between the concepts discussed in this paper and the (un)decidability of properties that we have obtained, and those used and obtained in the field of security protocols, like those studied in [10]. In particular, the study of the restrictions needed to preserve the decidability of coverability, could be related with some other recent studies in the field of security protocols [19, 20], while we could also try to use the efficient algorithms [1] that combine forward and backward reachability, to decide security properties of protocols.

References

[1] Abdulla, P.A., Deneux, J., Mahata, P., Nylén, A.: Forward Reachability Analysis of Timed Petri Nets. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 343–362. Springer, Heidelberg (2004)

[2] Ball, T., Chaki, S., Rajamani, S.K.: Parameterized Verification of Multithreaded Software Libraries. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 158–173. Springer, Heidelberg (2001)

[3] Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'03. ACM SIGPLAN, vol. 38(1), pp. 62-73. ACM (2003)

[4] Bouajjani, A., Esparza, J., Touili, T.: Reachability Analysis of Synchronized PA Systems. In: 6th Int. Workshop on Verification of Infinite-State Systems, INFINITY'04. ENTCS vol. 138(2), pp. 153-178. Elsevier, Amsterdam (2005)

[5] Busi, N., Zavattaro, G.: Deciding Reachability in Mobile Ambients. In: Sagiv, M. (ed.) ESTAPS'05. LNCS, vol. 3444, pp. 248–262. Springer, Heidelberg (2005)

[6] Cardelli, L., Gordon, A.D.: Mobile Ambients. In: Nivat, M. (ed.) ETAPS 1998 and FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)

[7] Cardelli, L., Ghelli, G., Gordon, A.D.: Types for the Ambient Calculus. Information and Computation 177(2), 160–194 (2002)

[8] Delzanno, G.: An overview of MSR(C): A CLP-based Framework for the Symbolic Verification of Parameterized Concurrent Systems. In: 11th Int. Workshop on Functional and Logic Programming, WFLP'02. ENTCS, vol. 76, Elsevier, Amsterdam (2002)

[9] Delzanno, G.: Constraint-based Automatic Verification of Abstract Models of Multitreaded Programs. To appear in the Journal of Theory and Practice of Logic Programming (2006)

[10] Durgin, N.A., Lincoln, P.D., Mitchell, J.C., Scedrov, A.: Undecidability of bounded security protocols. In: Proc. Workshop on Formal Methods and Security Protocols (FMSP'99)

[11] Finkel, A., Schnoebelen, P.: Well-Structured Transition Systems Everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001)

[12] Frutos-Escrig, D., Marroquín-Alonso, O., Rosa-Velardo, F.: Ubiquitous Systems and Petri Nets. In: Zhou, X., Li, J., Shen, H.T., Kitsuregawa, M., Zhang, Y. (eds.) APWeb 2006. LNCS, vol. 3841, Springer, Heidelberg (2005)

[13] Gordon, A.: Notes on Nominal Calculi for Security and Mobility. In: Focardi, R., Gorrieri, R. (eds.) Foundations of Security Analysis and Design (FOSAD'00). LNCS, vol. 2171, pp. 262–330. Springer, Heidelberg (2001)

[14] Kummer, O.: Undecidability in object-oriented Petri nets. Petri Net Newsletter 59, 18–23 (2000)

[15] Lazic, R.: Decidability of Reachability for Polymorphic Systems with Arrays: A Complete Classification. ENTCS 138(3), 3–19 (2005)

[16] Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I. Information and Computation 100(1), 1–40 (1992)

[17] Nielson, F., Hansen, R.R., Nielson, H.R.: Abstract interpretation of mobile ambients. Sci. Comput. Program 47(2-3), 145–175 (2003)

[18] Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)

[19] Ramanujam, R., Suresh, S.P.: Decidability of context-explicit security protocols. Journal of Computer Security 13(1), 135–165 (2005)

[20] Ramanujam, R., Suresh, S.P.: Tagging makes secrecy decidable with unbounded nonces as well. In: Pandya, P.K., Radhakrishnan, J. (eds.) FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 2914, pp. 363–374. Springer, Heidelberg (2003)

[21] Rinard, M.: Analysis of multithreaded programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 1–19. Springer, Heidelberg (2001)

[22] Rosa Velardo, F., Segura Díaz, C., de Frutos Escrig, D.: Tagged systems: a framework for the specification of history dependent properties. Fourth Spanish Conference on Programming and Computer Languages, PROLE'04. ENTCS vol. 137(1), (2005).

[23] Rosa-Velardo, F., Frutos-Escrig, D., Marroquín-Alonso, O.: Mobile Synchronizing Petri Nets: a choreographic approach for coordination in Ubiquitous Systems. In: 1st Int. Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord'05. ENTCS, vol. 150(1), Elsevier, Amsterdam (2006)

[24] Rosa-Velardo, F., Frutos-Escrig, D., Marroquín-Alonso, O.: On the expressiveness of Mobile Synchronizing Petri Nets. In: 3rd Int. Workshop on Security Issues in Concurrency, SecCo'05. ENTCS (to appear)

[25] Rosa-Velardo, F.: Coding Mobile Synchronizing Petri Nets into Rewriting Logic. 7th Int. Workshop on Rule-based Programming, RULE'06. ENTCS (to appear)

[26] Rosa-Velardo, F., Frutos-Escrig, D., Marroquín-Alonso, O.: Replicated Ubiquitous Nets. In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganà, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3983, Springer, Heidelberg (2006)

[27] Zimmer, P.: On the Expressiveness of Pure Mobile Ambients. In: 7th Int. Workshop on Expressiveness in Concurrency, EXPRESS'00. ENTCS, vol. 39(1), Elsevier, Amsterdam (2003)

Modelling the Datagram Congestion Control Protocol's Connection Management and Synchronization Procedures

Somsak Vanit-Anunchai* and Jonathan Billington

Computer Systems Engineering Centre
University of South Australia, Mawson Lakes Campus, SA 5095, Australia
vansy014@students.unisa.edu.au, jonathan.billington@unisa.edu.au

Abstract. The Datagram Congestion Control Protocol (DCCP) is a new transport protocol standardised by the Internet Engineering Task Force in March 2006. This paper specifies the connection management and synchronisation procedures of DCCP using Coloured Petri nets (CPNs). After introducing the protocol, we describe how the CPN model has evolved as DCCP was being developed. We focus on our experience of incremental enhancement and iterative modelling in the hope that this will provide guidance to those attempting to build complex protocol models. In particular we discuss how the architecture, data structures and specification style of the model have evolved as DCCP was developed. The impact of this work on the DCCP standard is also briefly discussed.

Keywords: Internet Protocols, DCCP, Coloured Petri Nets, Formal Specification.

1 Introduction

Background. Recently, the Internet Engineering Task Force (IETF) has published a set of standards for a new transport protocol, the Datagram Congestion Control Protocol (DCCP) [8], comprising four standards called Request For Comments (RFCs): RFC 4336 [2]; RFC 4340 [14]; RFC 4341 [3]; and RFC 4342 [4]. RFC 4336 discusses problems with existing transport protocols and the motivation for designing a new transport protocol. RFC 4340 specifies reliable connection management procedures; synchronisation procedures; and reliable negotiation of various options. RFC 4340 also provides a mechanism for negotiating the congestion control procedure to be used, called "Congestion Control Identification" (CCID). The congestion control mechanisms themselves are specified in other RFCs. Currently there are two published: RFC 4341 (CCID2) "TCP-like congestion control" [3] and RFC 4342 (CCID3) "TCP-Friendly Rate Control" [4] but more mechanisms are being developed for the standard.

* Partially supported by an Australian Research Council Discovery Grant (DP0559927).

Motivation. RFC 4340 [14] describes its procedures using narrative, state machines and pseudo code. The pseudo code has several drawbacks. Firstly, it does not include timers and user commands. Secondly, because it executes sequentially, unnecessary processes can execute before reaching the process that is needed. This unnecessary execution makes it difficult to read, which may lead to misinterpretation of the procedures. Thus to avoid misinterpretation, the implementor has to strictly follow the pseudo code. Following the sequential pseudo code, we cannot exploit the inherent concurrency in the protocol. Thirdly, there may also be subtle errors in the RFC which may be found using formal methods. This has motivated us to formally specify DCCP's procedures using Coloured Petri Nets (CPNs) [7].

Previous Work. We have closely followed the development, design and specification of DCCP Connection Management (DCCP-CM) since Internet Draft version 5 [16]. Our study aims to determine if any errors or deficiencies exist in the protocol procedures. We analysed the connection establishment procedures of version 5 and discovered a deadlock [19]. When version 6 [9] was released, we upgraded our CPN model [17] and found similar deadlocks. Version 11 of the DCCP specification was submitted to IETF for approval as a standard. We updated our model accordingly and included the synchronization mechanism. We found that the deadlocks were removed but we discovered a serious problem known as *chatter*[1] [23,21]. When updating the CPN model to RFC 4340, we investigated the connection establishment procedure when sequence numbers wrap [20]. We found that in this case it is possible that the attempt to set up the connection fails. In [5] we defined the DCCP service and confirmed that the sequence of user observable events of the protocol conformed to its service. In [22] we proposed an idea to improve the efficiency of the sweep-line method using DCCP as a case study.

However, all of these papers place emphasis on the model's analysis and present very little of the model itself. In contrast, this paper focuses on the modelling process. It discusses design decisions and how the DCCP-CM CPN models were developed rather than the analysis of the protocol procedures. Our final version of the DCCP-CM CPN model is the result of incremental refinement of three earlier versions which were used to obtain the results in [19,17,23].

Contribution. We use Coloured Petri Nets to model the DCCP connection management and synchronization procedures of RFC 4340. In contrast to the pseudo code, our proposed CPN specification includes user commands, timer expiry and channels. It is formalised so that ambiguities are removed and the risk of misinterpretation is low. The specification is restructured so that relevant aspects of the procedures are brought together and don't need to be sought in different parts of the document. It now captures concurrent behaviour which is inherent in the protocol whereas the pseudo code and state machine are sequential. The CPN model can be analysed to determine errors. In particular we

[1] The chatter scenario comprised undesired interactions between Reset and Sync packets involving a long but finite exchange of messages where no progress was made, until finally the system corrected itself.

illustrate how to use hierarchical CPNs to model DCCP connection management and discuss our experience during the modelling process.

The rest of the paper is organised as follows. Section 2 provides an overview of DCCP and its connection management procedures. Section 3 describes the evolution of our DCCP-CM CPN model as DCCP has been developed by IETF. Finally, conclusions and future work are discussed in Section 4.

2 Overview of DCCP's Connection Management Procedures

DCCP is a point-to-point protocol operating over the Internet between two DCCP entities, which we shall call the client and the server. DCCP is connection-oriented to facilitate congestion control. DCCP's procedures for connection management comprise connection establishment and five closing procedures. RFC 4340 includes the state transition diagram shown in Fig. 1(a), which gives the basic procedures for both the server and client entities. The server is identified by always passing through the LISTEN state after a "passive open" command from the application, whereas the client is instantiated by an "active open" and passes through the REQUEST state. Also, only the server can issue the "server active close" command.

2.1 DCCP Packet Format

A set of messages, called packets, is used to setup and release connections, synchronize sequence numbers, and to convey data between a client and a server. RFC 4340 [14] defines a DCCP packet as a sequence of 32 bit words comprising a DCCP Header and Application Data as shown in Fig. 1(b). The header comprises a generic header, followed by an acknowledgement number (if any) and then fields for options and features. The DCCP header contains 16 bit source and destination port numbers (for addressing application processes), and a 16 bit checksum to detect transmission errors. Since the Options field can vary in length, an 8 bit data offset indicates the length in 32-bit words from the beginning of the Header to the beginning of the Application data. CCVal, a 4 bit field, is a value used by the congestion control mechanisms [4]. Checksum Coverage (CsCov), also a 4 bit field, specifies the part of the packet being protected by the checksum. A four bit field specifies the type of packet: Request, Response, Data, DataAck, Ack, CloseReq, Close, Reset, Sync and SyncAck. By default, packets include a 48-bit sequence number. Request and Data packets do not include acknowledgement numbers. The sequence numbers of Data packets and the sequence numbers and acknowledgement numbers of Ack and DataAck packets can be reduced to 24-bit (*short*) when setting the Extend Sequence Number (X) field to 0. The Option fields contain state information or commands for applications to negotiate various features such as the Congestion Control Identifier (CCID) and the width of the sequence number validity window [14].

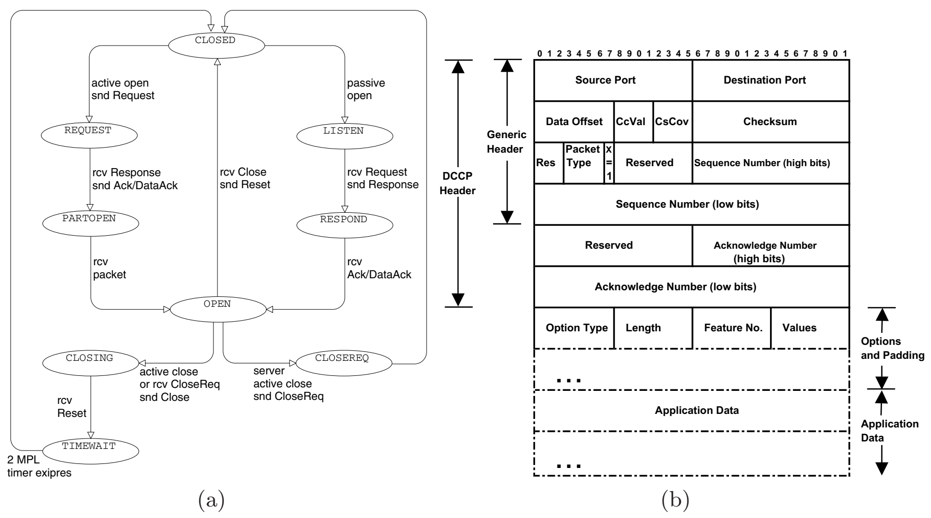


Fig. 1. (a) DCCP state diagram redrawn from [14] (b) DCCP packet format

2.2 DCCP Connection Management Procedures

The typical procedure of connection establishment is shown in the time sequence diagram of Fig. 2(a). We assume that the server has entered the LISTEN state after a "passive open" command is issued by the application at the server. On receiving an "active open" command from its application, the client enters the REQUEST state and sends a DCCP-Request packet requesting a connection to be established. This packet may include any options and features that require negotiation.

When receiving the DCCP-Request packet, the server (in LISTEN) enters the RESPOND state, and replies with a DCCP-Response packet indicating its willingness to establish the connection. The DCCP-Response will also include any options and features being negotiated. On receiving the DCCP-Response, the client (in REQUEST) replies with either a DCCP-Ack or DCCP-DataAck packet to acknowledge the DCCP-Response and enters PARTOPEN. After receiving an acknowledgement from the client, the server enters the OPEN state and may start transferring data. After receiving a DCCP-Data, DCCP-Ack or DCCP-DataAck packet, the client enters OPEN indicating that the connection has been established. During data transfer, the server and client may exchange DCCP-Data, DCCP-Ack and DCCP-DataAck packets.

In order to ensure that no old packets are delivered in the new connection, one side has to wait for 2 Maximum packet lifetimes (MPL[2]) before a connection can be restarted. To improve performance, especially in the face of state exhaustion attacks[3], the server should not be burdened with this delay. Typically it is the

[2] Maximum packet lifetime time (MPL) = Maximum Segment Lifetime (MSL) in TCP.

[3] A state exhaustion attack involves attackers trying to use up the server's memory by requesting an enormous number of connections to be established.

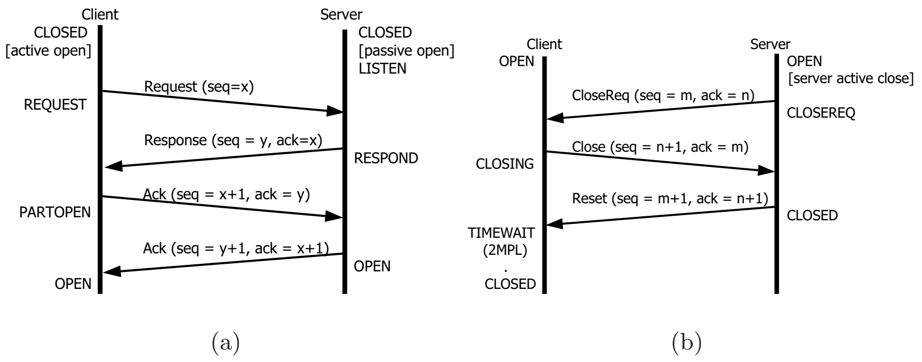


Fig. 2. Typical connection establishment and release scenarios

client that waits before closing is complete. Fig. 2(b) shows the typical close down procedure when the application at the server initiates connection release by issuing a "server active close" command. After sending a DCCP-CloseReq packet, the server enters the CLOSEREQ state. On receiving a DCCP-CloseReq, the client replies with a DCCP-Close to which the server responds with a DCCP-Reset packet and enters the CLOSED state. On receiving the DCCP-Reset, the client waits in the TIMEWAIT state for 2 MPL before entering the CLOSED state.

Either entity may send a DCCP-Close packet to terminate the connection on receiving an "active close" command from the application. The entity that receives the DCCP-Reset packet will enter the TIMEWAIT state. There are another 2 possible scenarios beside these three closing procedures called *simultaneous closing*. One scenario occurs when applications on both sides issue an "active close". The other is when the client user issues an "active close" and the server user issues the "server active close" command.

2.3 Synchronization Procedure

During the connection, DCCP entities maintain a set of state variables. The important variables are Greatest Sequence Number Sent (GSS), Greatest Sequence Number Received (GSR), Greatest Acknowledgement Number Received (GAR), Initial Sequence Number Sent and Received (ISS and ISR), Valid Sequence Number window width (W) and Acknowledgement Number validity window width (AW). Based on the state variables, the valid sequence and acknowledgement number intervals are defined by Sequence Number Window Low and High [SWL,SWH], and Acknowledgement Number Window Low and High [AWL,AWH] (see [14,23,20]). In general, received packets having sequence and acknowledgement numbers outside these windows are *sequence-invalid*. DCCP performs a sequence validity check when packets arrive but not in the CLOSED, LISTEN, REQUEST and TIMEWAIT states.

On receiving a sequence-invalid packet, DCCP re-synchronizes the state variables, GSR and GAR, by sending a DCCP-Sync packet. It does not update its GSR and GAR. According to section 7.5.4 of [14] the acknowledgement number in the DCCP-Sync is set equal to the invalid received sequence number (except when receiving an invalid DCCP-Reset). After the other end receives the DCCP-Sync, it updates GSR and replies with a DCCP SyncAck. It does not update GAR since the acknowledge number is sequence-invalid. Upon receiving the DCCP-SyncAck, the DCCP entity updates GSR and GAR. A sequence-invalid DCCP-Sync and DCCP-SyncAck must be ignored. However if the entity receives a DCCP-Sync or DCCP-SyncAck (either valid or invalid) in CLOSED, LISTEN, REQUEST and TIMEWAIT, it replies with a DCCP-Reset packet.

3 CPN Model Development

This section describes our CPN model and how it has evolved during the period of modelling as DCCP was developed in a series of Internet Drafts. We start with a simple and abstract model with a lot of assumptions. While the DCCP specification was being developed by IETF, our DCCP-CM CPN models have been gradually refined to faithfully reflect the DCCP specification. Our contribution to RFC 4340 is also discussed in this section.

Our models were created and maintained using Design/CPN [18], a software package for the creation, editing, simulation and state space analysis of CPNs. Design/CPN supports the hierarchical construction of CPNs [7], using constructs called *substitution transitions*. These transitions (macro expansions) hide the details of subnets and allow further nesting of substitution transitions. This allows a complex specification to be managed as a series of hierarchically related pages which are visualised in a *hierarchy page*, automatically generated by Design/CPN.

3.1 Refinement of the Hierarchical Structure of DCCP-CM CPN Model

The first step of modelling DCCP connection management (DCCP-CM) is to identify the interaction between DCCP entities and their environment. The top level of our initial CPN model [19] (Internet Draft version 5) shown in Fig. 3 comprises six places and two substitution transitions. Substitution transition DCCP_C represents the client and DCCP_S represents the server. There are two places, App_Client and App_Server which store application commands (e.g. Active Open), for the client and server, two places, Client_State and Server_State, that store DCCP entity states, including state variables, in a so called Control Block (CB) and two places, CH_C_S and CH_S_C, representing channels between the client and server (one for each direction). The layout of the model is suggestive of the conceptual locations of the entities being modelled: the applications in the layer "above" DCCP; the client on the left and server on the right; and the channels between the client and server. This follows the practice adopted

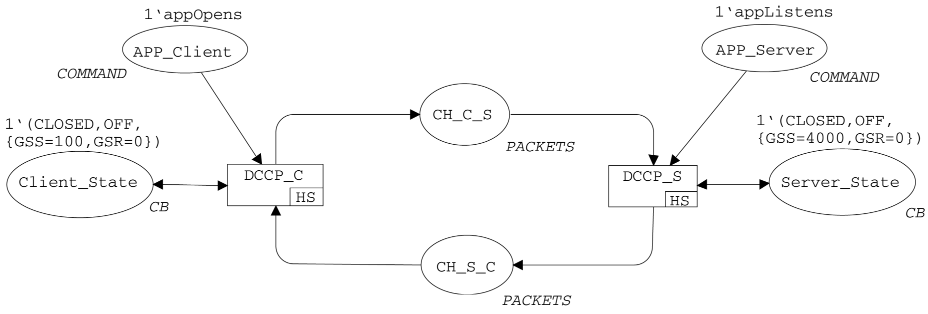


Fig. 3. The top level page from [19]

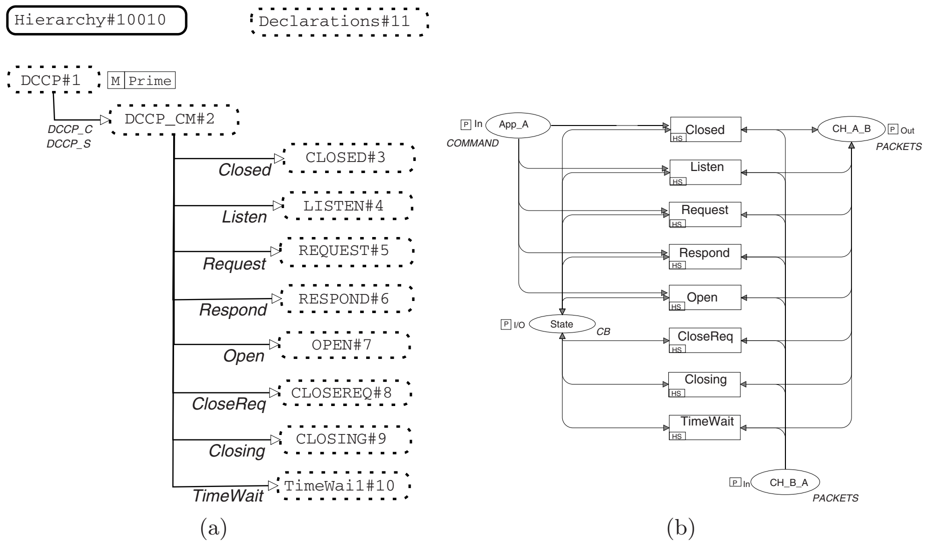


Fig. 4. (a) The hierarchy page from [19] (b) The second level page from [19]

in [1] and is very similar to the top level page of our CPN model of TCP [1,6]. We initially assume the packets in channels can be reordered but not lost.

The next step is to model the DCCP entities in the second level CPN page. The substitution transitions DCCP_C and DCCP_S are linked to the DCCP_CM#2 page (see Fig. 4 (a)). The DCCP_CM#2 is arranged into a further eight[4] substitution transitions (Fig. 4 (b)) named after DCCP major states. Each substitution transition is linked to a third level CPN page which models DCCP behaviour for each major state.

This modelling method follows a "state-based" approach because the model is similar to how DCCP is specified using the state diagram. In a state-based approach, every major state is considered in turn, and in each state, transitions

[4] DCCP Internet Draft version 5 did not include the PARTOPEN state.

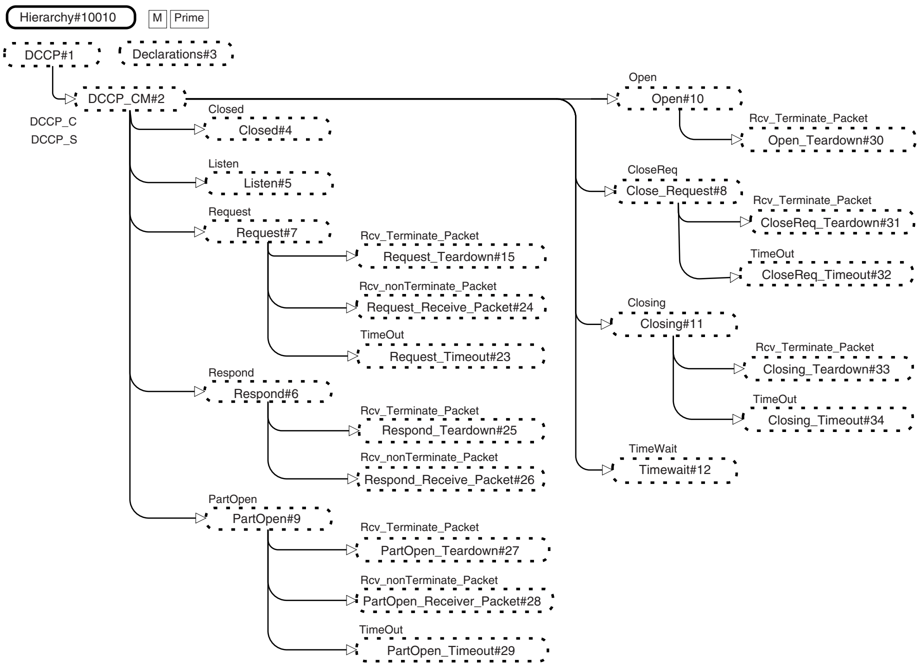


Fig. 5. The hierarchy page from [17]

are created for each input, which may be a user command, the arrival of a packet or the expiry of a timer. The state-based approach is very suitable when we are creating and building a new model as it is easy to understand and validate against the state machine and the narrative description. A disadvantage of the state-based approach is that the model may include a lot of redundancy because different states may process the same inputs in the same or similar way.

When we upgraded the model [17] to DCCP version 6, one state could have many executable transitions (more than 7-8 transitions). Thus we regrouped transitions that have similar functions into fourth level pages as shown in Fig 5. For example, the client's actions in the REQUEST state are rearranged into three groups: terminating the connection; not-terminating the connection; and retransmission on time-out.

As we refined the state-based CPN model by including more detail of the Internet Drafts, it became increasingly difficult to maintain the model. We faced this difficulty when developing the CPN model [23] for version 11. To overcome this problem we folded transitions that are common to many states and moved them to a CommonProcessing#7 page as shown in Fig. 6. The obvious common actions are how to respond to DCCP-Sync and DCCP-Reset packets. This resulted in an immediate pay off. When we reported the chatter problem [23] in version 11 [11] to IETF, IETF asked us to test their solution [13] in our CPN model. To incorporate the solution we changed only a single variable on an arc in the Reset_Rcv page.

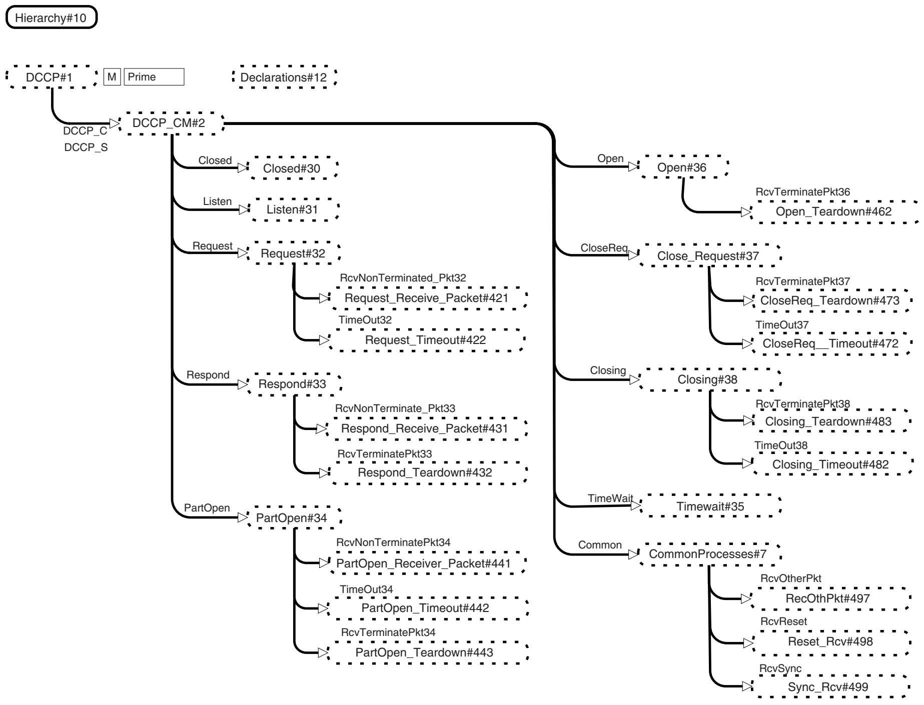


Fig. 6. The hierarchy page from [23]

We could thus analyse the modified model within five minutes to show that chatter had been removed. The important merit of removing redundancy by folding similar transitions is the ability to maintain the model efficiently. The resulting CPN model is a little more difficult to read, but the maintenance pay off is well worth it.

When the specification is mature and relatively stable, we recommend removing redundancies by folding similar transitions so that it is easier to maintain. After IETF had revised DCCP and issued version 13 [12], we decided to remove further redundancies by folding and regrouping all transitions (which we illustrate in section 3.3). During the folding process we considered a trade-off between redundancy and readability. The structure of the final CPN model is shown in Fig. 7. We merged the CLOSED, LISTEN and TIMEWAIT states and gathered them into the IdleState page. The procedures that are common to states other than the idle states were regrouped into new pages, including the UserCommands and ClosingDown pages, and those grouped together under the CommonProcessing page such as the Retransmission page. This also involved some renaming of pages and resulted in a much simpler architecture as can be seen in Fig. 7. The number of pages was reduced from 32 to 15, and the number of transitions from 86 to 52.

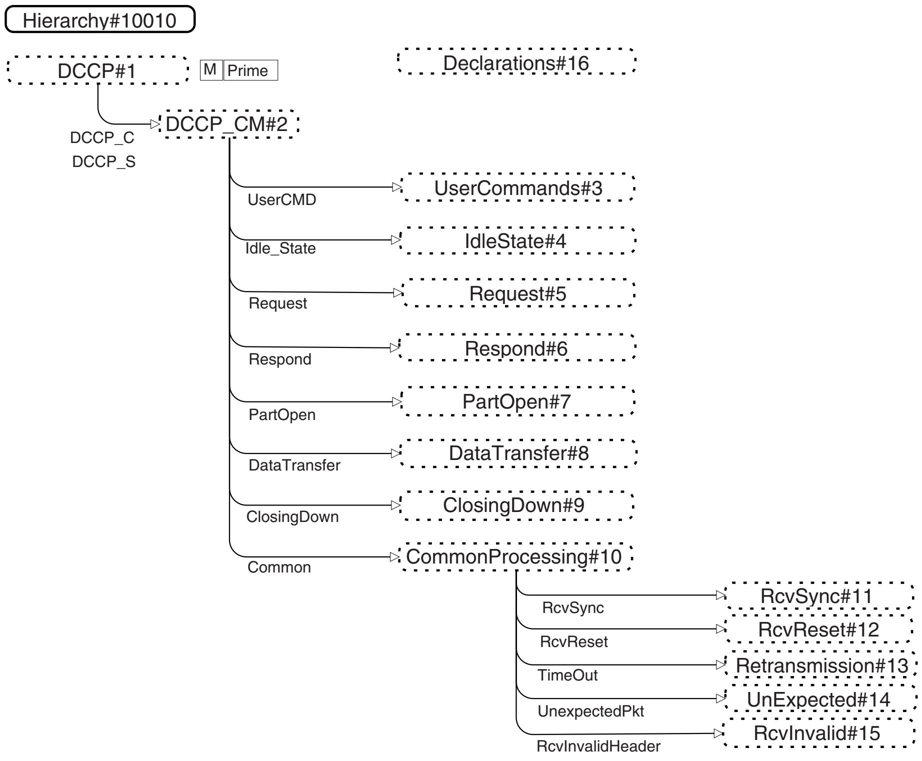


Fig. 7. The hierarchy page of the final DCCP-CM CPN model

3.2 Refinement of the Data Structures of the DCCP-CM CPN Model

In the previous subsection we discussed the modelling process and the refinement of the structure of our models. This section discusses the way we refined the data structures over four versions of DCCP-CM CPN model [19,17,23,22]. The DCCP-CM model we initially built and published in [19] is based on DCCP version 5 [16]. The model was simple and abstract and has the following merits: compact, easy to understand, rapidly built and fast to analyse. It took only two weeks for an inexperienced student to create and test the model [19]. A deadlock was quickly discovered using state space analysis.

DCCP Internet Draft Version 5. Our initial model [19] was based on a number of assumptions:

1) A DCCP packet was modelled by its packet type and sequence and acknowledgement numbers. Other fields in the DCCP header were omitted because they do not affect the operation of the connection management procedure.

```
color STATE = with CLOSED | LISTEN | REQUEST | RESPOND | OPEN | CLOSEREQ
                | CLOSING | TIMEWAIT;
color BackOffFlag = with ON | OFF;
color StateVariable = record GSS:INT*GSR:INT;
color CB = product STATE*BackOffFlag*StateVariable;
color PacketType = with Request | Response | Ack | DataAck | Data | CloseReq
                                | Close | Rst ;
color SeqAck  = record SEQ:INT*ACK:INT;
color PACKETS = product PacketType*SeqAck;
```

Fig. 8. Definition of DCCP's Control Block and PACKETS from [19]

2) DCCP version 5 specified 24-bit sequence numbers and provided an option for extended 48-bit sequence numbers. We only modelled 24-bit sequence numbers.

3) Version 5 specified that the Request and Data packets do not have acknowledgement numbers. However for ease and speed of creating the model we allowed these two packet types to have dummy acknowledgement numbers.

4) State variable GAR was omitted because it was optional in DCCP version 5.

5) After version 5 was released, there was a lot discussion in IETF regarding the synchronization procedure. At that time the synchronization procedure was not mature. (For example version 5 did not specify the DCCP-SyncAck packet.) Hence synchronization was not included.

6) Processing of unexpected packets was not taken into account.

7) Wrapping of sequence numbers was not included.

8) Features/options were not included.

9) Malicious attacks were not considered.

Assumptions 3 and 4 allowed us to defined the CB (Control Block) in Fig. 8 to be a product of STATE, BackOffFlag and a record of GSS and GSR. For example, Fig. 3 shows the initial marking 1'(CLOSED, OFF, {GSS=100, GSR=0}) in the place Client_State. Because we used the same data structure for every state, the CPN model was simple and built rapidly. As a result of assumption 7, we did not need to implement modulo arithmetic, however, our results were restricted to when sequence numbers do not wrap. Despite all these assumptions, our analysis of the initial DCCP-CM CPN model revealed that a deadlock occurred during connection setup [19]. The deadlock was due to Reset packets with sequence number zero being valid. Hence the deadlock we found was not due to any of the assumptions made.

DCCP Internet Draft Version 6. When we updated the DCCP-CM model to DCCP version 6 [17], we attempted to faithfully define the packet structure. Firstly, we restricted the value of the sequence number to range from zero to $2^{24} - 1$ (line 1 of Fig. 9). Secondly, the sequence and acknowledgement number record (SeqAck) was redefined as a union of the sets SeqWAck and SEQ24 (line 5 of Fig. 9). Thus the Request and Data packets, which do not include an acknowledgement number, can be modelled accurately. The model in [17] also includes the PARTOPEN state but does not include the re-synchronization process. The analysis of the CPN model of DCCP Internet Draft version 6 showed the deadlock problem was still present.

```
color SEQ24  = int with 0..16777215; (* 24-bit sequence number *)
color PacketType = with Request | Response | Ack | DataAck | Data | CloseReq
                        | Close | Rst;
color SeqWAck  = record SEQ:SEQ24*ACK:SEQ24;
color SeqAck   = union WAck:SeqWAck + SEQ24;
color PACKETS = product PacketType*SeqAck;
```

Fig. 9. Definition of PACKETS from [17]

```
1: color PacketType1 = with Request | Data;
2: color PacketType2 = with Sync | SyncAck | Response | Ack | DataAck
3:                   | CloseReq | Close | Rst;
4: color SN = IntInf with ZERO..MaxSeqNo;
5: color SN_AN = record SEQ:SN*ACK:SN;
6: color PacketType1xSN = product PacketType1*SN;
7: color PacketType2xSN_AN = product PacketType2*SN_AN;
8: color PACKETS = union PKT1:PacketType1xSN+PKT2:PacketType2xSN_AN;
```

Fig. 10. Definition of PACKETS from [23]

DCCP Internet Draft Version 11. Instead of 24-bit sequence numbers, Internet Draft version 7 [10] specified 48-bit long sequence numbers as mandatory for all packet types. Short sequence numbers were only allowed to be used in Data, Ack and DataAck packets. We upgraded the DCCP-CM CPN model [23] according to DCCP version 11 [11]. This model represents long sequence numbers by "infinite integers" in ML, ranging from zero to $2^{48} - 1$ (line 4 of Fig. 10) but does not include short sequence numbers. We also incorporated an up-to-date algorithm for checking sequence number validity and included the synchronization mechanism. The previous declaration in Fig. 9 allows Request and Data packets to have acknowledgement numbers and other packet types to be without acknowledgement numbers. To improve the representation of the packet structure, we defined PacketType1 to be a DCCP packet without an acknowledgement number and PacketType2 to be a DCCP packet with an acknowledgement number as shown in Fig. 10 (line 1-2).

In DCCP versions 5 and 6 we found deadlocks because a delayed Reset packet with a sequence number of zero is always valid. Since version 7 [10] a Reset packet with a sequence number of zero was no longer always valid. This modification to the specification removed these deadlock problems. Although we did not find any deadlocks or livelocks in DCCP version 11, we discovered chatter [23]. In [21] we showed that if a chattering scenario occurs, it is highly likely (probability > 0.99999) that more than two billion messages will be needlessly exchanged. Although the chance of a particular scenario is low, we discovered a large number of possible chattering scenarios.

The Internet Standard RFC4340. After submitting the discovery of chatter to IETF, the principal editor of the specification advised us that the chatter problem was more severe than we originally thought, especially when using short sequence numbers. This motivated us to revise the model to include both long and short sequence numbers. Figure 11 shows the definition of PACKETS in the final CPN model. As before, long sequence numbers (SN48) in line 15 of Fig. 11 are

```
 1: (* Packet Structure *)
 2: (* Maximum Sequence Number Values *)
 3: val MaxSeqNo48 = IntInf.-(IntInf.pow(IntInf.fromInt(2),48),IntInf.fromInt(1));
 4: val MaxSeqNo24 = IntInf.-(IntInf.pow(IntInf.fromInt(2),24),IntInf.fromInt(1));
 5: val max_seq_no24 = IntInf.toInt(MaxSeqNo24);
 6: (* Packet Types     *)
 7: color PktType1 = with Request | Data;
 8: color PktType2 = with Sync | SyncAck | Response | Ack | DataAck
 9:                 | CloseReq | Close | Rst;
10: color DATA=subset PktType1 with [Data];
11: color ACK_DATAACK=subset PktType2 with [DataAck,Ack];
12: (* Extended Sequence Number Flag *)
13: color X = with LONG | SHORT;
14: (* Sequence Number *)
15: color SN48 = IntInf with ZERO..MaxSeqNo48;
16: color SN48_AN48 = record SEQ:SN48*ACK:SN48;
17: color SN24 = int with 0..max_seq_no24;
18: color SN24_AN24 = record SEQ:SN24*ACK:SN24;
19: (* Four Different Types of Packets *)
20: color Type1LongPkt=product PktType1*X*SN48;
21: color Type2LongPkt=product PktType2*X*SN48_AN48;
22: color Type1ShortPkt=product DATA*X*SN24;
23: color Type2ShortPkt=product ACK_DATAACK*X*SN24_AN24;
24: color PACKETS=union PKT1:Type1LongPkt   + PKT2:Type2LongPkt
25:                   + PKT1s:Type1ShortPkt + PKT2s:Type2ShortPkt;
26: (* Variables *)
27: var sn:SN48;   var sn_an:SN48_AN48;
28: var sn24:SN24; var sn24_an24:SN24_AN24;
29: var LS:BOOL;   var ack_dataack:ACK_DATAACK;
```

Fig. 11. Definition of DCCP PACKETS in the final CPN model

represented by infinite integers ranging from zero to $2^{48}-1$. Short sequence numbers (SN24) in line 17 are represented by integers ranging from zero to $2^{24}-1$. The strong typing of packets is very useful for developing and debugging the model, as certain mistakes can be easily detected by a syntax check.

While analysing the revised model, we found an error in the algorithm for extending short sequence numbers to long sequence numbers (page 60 of [11]) and submitted the problem to IETF. IETF devised solutions to these problems (chatter and the algorithm for short sequence number extension) [13]. These solutions have been incorporated into RFC 4340 [14].

When revising the CPN model, we attempted to define the control block in Fig. 12 so that it truly reflected DCCP's definition and removed ambiguities. When in CLOSED and LISTEN, the GSS, GSR, GAR, ISS and ISR state variables do not exist, while the client in the REQUEST state has only instantiated GSS and ISS. Thus we classify DCCP states into three groups: idle, request and active states. These three groups also have differences regarding the functional behaviour of how to respond to the DCCP-Reset and DCCP-Sync packets. We suggested to IETF that it is not necessary for TIMEWAIT to maintain state variables because the connection is about to close. It was agreed that TIMEWAIT should be classified in the group of idle states [15]. This helps to reduce the size of the state space (e.g. from 42,192 to 26,859 nodes for the case of connection establishment).

We define each group of states with a different set of state variables. Fig. 12 defines CB (line 14) as a union of colour sets: IDLE, REQUEST and

```
 1: (* DCCP state variables*)
 2: color RCNT=int;(*Retransmission Counter*)
 3: (* FSM State *)
 4: color IDLE=with CLOSED_I| LISTEN| CLOSED_F| TIMEWAIT;
 5: color REQUEST = product RCNT*SN48*SN48;
 6: color ACTIVE = with RESPOND| PARTOPEN| S_OPEN| C_OPEN
 7:                 | CLOSEREQ | C_CLOSING |S_CLOSING;
 8: (* Sequence Number Variables *)
 9: color GS = record GSS:SN48*GSR:SN48*GAR:SN48;
10: (* Initial Sequence Number *)
11: color ISN = record ISS:SN48*ISR:SN48;
12: (* Control Block *)
13: color ACTIVExRCNTxGSxISN=product ACTIVE*RCNT*GS*ISN;
14: color CB = union IdleState:IDLE
15:                + ReqState:REQUEST
16:                + ActiveState:ACTIVExRCNTxGSxISN;
17: (* User Commmands *)
18: color COMMAND = with p_Open | a_Open | a_Close | server_a_Close;
19: (* Variables *)
20: var idle_state:IDLE; var active_state:ACTIVE; var g:GS;
```

Fig. 12. DCCP's control block and user commands in the final model

ACTIVExRCNTxGSxISN. IDLE (line 4) defines three idle states: CLOSED, LISTEN and TIMEWAIT. The CLOSED state is split into CLOSED_I to represent the initial CLOSED state and CLOSED_F to represent a terminal CLOSED state. This separation is useful when analysing state spaces [22]. The colour set REQUEST is a product comprising RCNT (Retransmission Counter, line 2), GSS and ISS. Because there is only one state in this group, the REQUEST state is already distinguished from other states by using the ML selector ReqState in the union (line 14). The ACTIVExRCNTxGSxISN (line 13) is a product comprising ACTIVE (line 6), RCNT, GS (Greatest Sequence and Acknowledgement Numbers, line 9) and ISN (Initial Sequence Numbers, line 11). ACTIVE (line 6) defines five DCCP states: RESPOND, PARTOPEN, OPEN, CLOSEREQ and CLOSING. Because the client and server respond to the CloseReq packet differently in the OPEN and CLOSING states, we differentiate these states for the client and server: C_OPEN and C_CLOSING for the client; and S_OPEN and S_CLOSING for the server.

Figure 12 also defines the colour set, COMMAND, on line 18. The places App_Client and App_Server in Fig. 3, typed by COMMAND, model DCCP user commands (i.e. commands that can be issued by the applications that use DCCP). For example, the user command 1'a_Open is the initial marking of App_Client indicating that the client's application desires to open a connection.

3.3 Illustration of Folding and Regrouping of Transitions at the Executable Level

This section discusses some examples of how the DCCP_CM CPN model has evolved at the executable level from the purely state-based approach to a mixed state-based and event processing model. We firstly present the second-level page to provide an overview of the revised model.

The Second Level Page. Figure 13 shows the DCCP_CM page which comprises eight substitution transitions. UserCMD models the actions taken when receiving commands from users. Idle_State combines similar processing actions required in the CLOSED, LISTEN and TIMEWAIT states. Request, Respond and PartOpen encapsulate the major processing actions in each of the corresponding states. DataTransfer represents actions taken when receiving the DCCP-Data, DCCP-Ack and DCCP-DataAck packets in the OPEN, CLOSEREQ and CLOSING states. ClosingDown defines the procedures undertaken on receiving a DCCP-Close or DCCP-CloseReq packet in the five active states: RESPOND, PARTOPEN, OPEN, CLOSEREQ and CLOSING. Finally, Common Processing comprises procedures that are common to various states including packet retransmissions, timer expiry, and the receipt of DCCP-Reset, DCCP-Sync and DCCP-SyncAck packets, unexpected packets and packets with an invalid header.

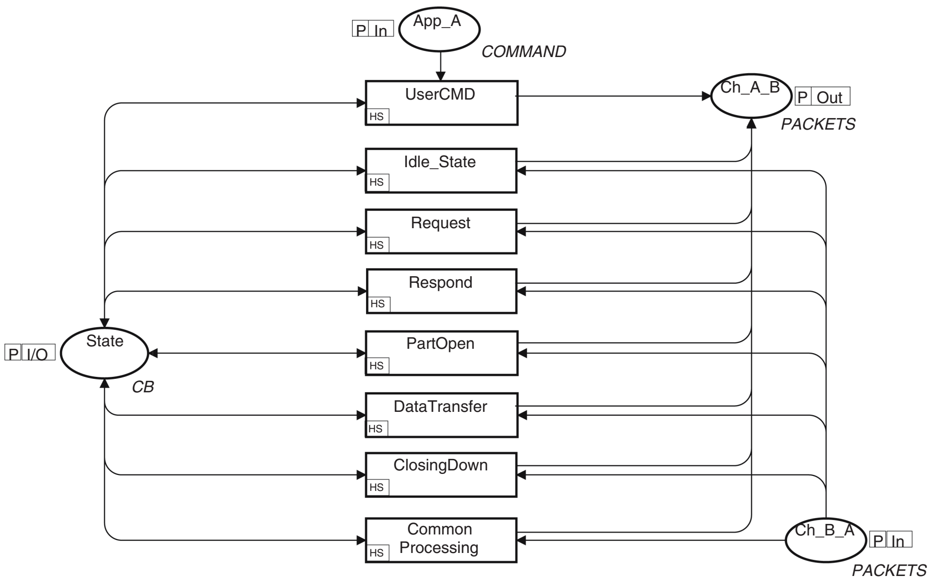


Fig. 13. The DCCP_CM page

The IdleState Page. The "state-based" model of Fig. 6 was further refined to include the processing of short sequence numbers. The resulting procedures for DCCP in the CLOSED, LISTEN and TIMEWAIT states are shown in Figs 14, 15 and 16. It is apparent that the procedures for receiving packets in each of these three states are almost identical. The only difference is the receipt of the Request packet in LISTEN. We also observe that the TIMEWAIT state includes state variables such as those required for recording sequence number values (e.g. for GSS, GSR and GAR). However, on close inspection we considered that these variables were not required. We discussed this issue with the main IETF DCCP editor, who agreed to remove ambiguity in the specification to

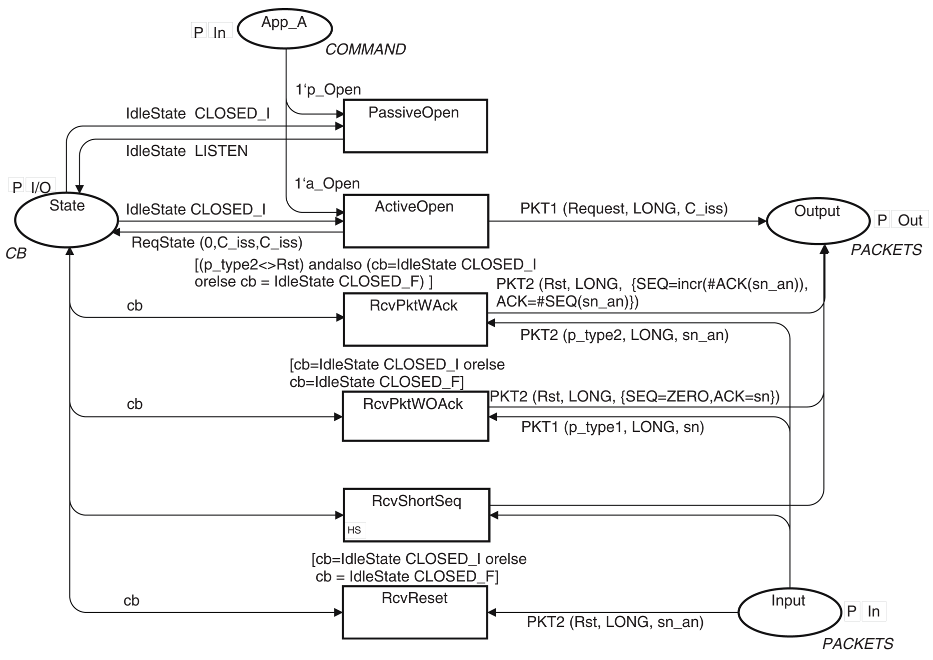

App_A
P In
COMMAND
1‘p_Open
PassiveOpen
IdleState CLOSED_I
IdleState LISTEN
P I/O
State
CB
1‘a_Open
ActiveOpen
IdleState CLOSED_I
ReqState (0,C_iss,C_iss)
PKT1 (Request, LONG, C_iss)
Output
P Out
PACKETS
[(p_type2<>Rst) andalso (cb=IdleState CLOSED_I orelse cb = IdleState CLOSED_F)]
cb
RcvPktWAck
PKT2 (Rst, LONG, {SEQ=incr(#ACK(sn_an)), ACK=#SEQ(sn_an)})
PKT2 (p_type2, LONG, sn_an)
[cb=IdleState CLOSED_I orelse cb=IdleState CLOSED_F]
cb
RcvPktWOAck
PKT2 (Rst, LONG, {SEQ=ZERO,ACK=sn})
PKT1 (p_type1, LONG, sn)
RcvShortSeq
HS
[cb=IdleState CLOSED_I orelse cb = IdleState CLOSED_F]
cb
RcvReset
PKT2 (Rst, LONG, sn_an)
Input
P In
PACKETS

Fig. 14. The CLOSED page

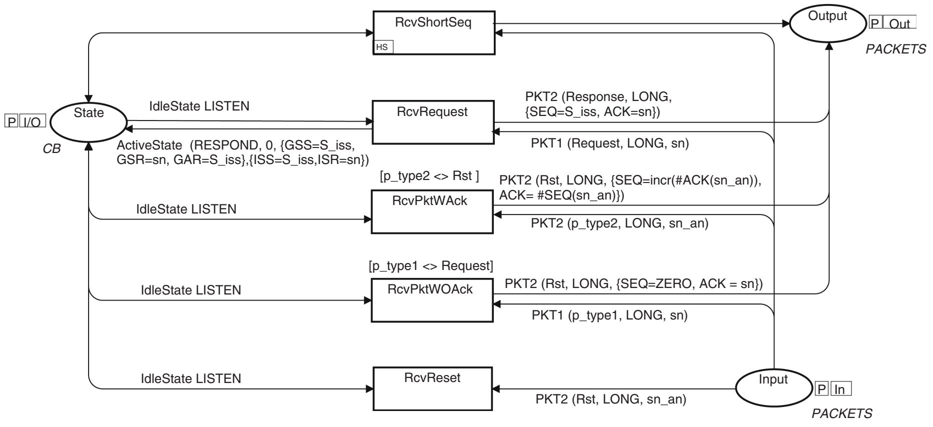


Fig. 15. The LISTEN page

make it clear that these variables are not needed in TIMEWAIT. We therefore deleted these variables. Further we grouped all the user commands together in a new page called UserCommands, thus removing transitions PassiveOpen and ActiveOpen from the CLOSED page. This then allowed the CLOSED, LISTEN and TIMEWAIT pages to be merged into a single page called IdleState shown in Fig. 17. The RcvRequest and TimerExpire transitions remain the same, but the

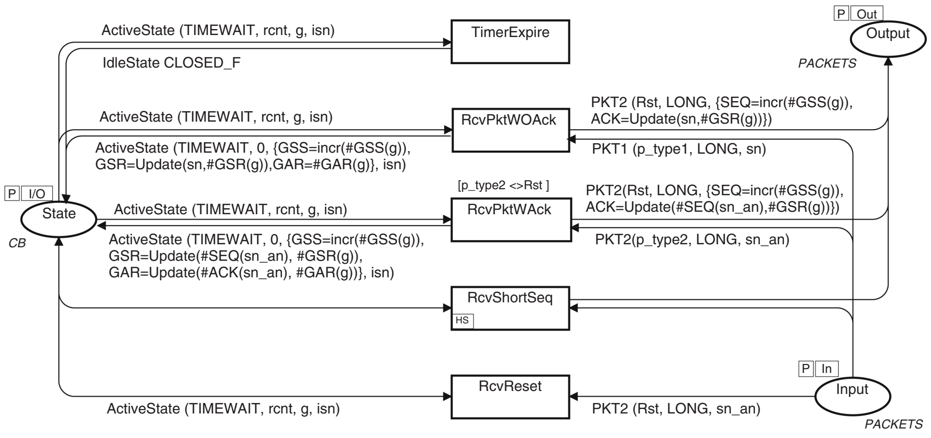

P Out
Output
PACKETS
ActiveState (TIMEWAIT, rcnt, g, isn)
TimerExpire
IdleState CLOSED_F
PKT2 (Rst, LONG, {SEQ=incr(#GSS(g)), ACK=Update(sn,#GSR(g))})
ActiveState (TIMEWAIT, rcnt, g, isn)
RcvPktWOAck
ActiveState (TIMEWAIT, 0, {GSS=incr(#GSS(g)), GSR=Update(sn,#GSR(g)),GAR=#GAR(g)}, isn)
PKT1 (p_type1, LONG, sn)
[p_type2 <>Rst]
P I/O
State
CB
ActiveState (TIMEWAIT, rcnt, g, isn)
RcvPktWAck
PKT2(Rst, LONG, {SEQ=incr(#GSS(g)), ACK=Update(#SEQ(sn_an),#GSR(g))})
ActiveState (TIMEWAIT, 0, {GSS=incr(#GSS(g)), GSR=Update(#SEQ(sn_an), #GSR(g)), GAR=Update(#ACK(sn_an), #GAR(g))}, isn)
PKT2(p_type2, LONG, sn_an)
RcvShortSeq
HS
P In
Input
RcvReset
ActiveState (TIMEWAIT, rcnt, g, isn)
PKT2 (Rst, LONG, sn_an)
PACKETS

Fig. 16. The TIMEWAIT page

other transitions for receipt of packets are folded together, and the substitution transition for the processing of short sequence number packets is expanded, so that all processing at a similar level is included on this page. This reduces the number of (executable) transitions from 17 to 7, and removes 3 substitution transitions. The folded model thus elegantly captures common processing in these states, highlights the differences and reduces maintenance effort, at the cost of introducing a variable that runs over the idle states.

The RcvReset Page. During the development process we gained insight into the protocol's behaviour. An example of this is that DCCP's behaviour, when receiving a Reset packet, can be classified into three cases. Firstly, the CLOSED, LISTEN and TIMEWAIT states ignore Reset packets as shown in Fig. 17. Secondly the client in the REQUEST state replies with a Reset packet. Because the client in REQUEST has not yet recorded an ISR, the acknowledgement number of the outgoing Reset is equal to zero according to section 8.1.1 of [14]. Thirdly, when the DCCP entity in any other active state receives a valid Reset, it enters TIMEWAIT but on receiving an invalid Reset, it replies with a Sync packet. However the Sync packet has an acknowledgement number equal to GSR rather than the sequence number received (section 7.5.4 of [14]). Instead of including a RcvReset transition in every state page, we group this behaviour into one CPN page as shown in Fig. 18. It comprises three transitions: InActiveState, InRequestState and InIdleState. Each one models the action of DCCP when receiving a Reset packet in a state corresponding to its name. Thus we moved transition RcvReset from IdleState to the RcvReset page. This page illustrates the "event processing" style of specification. In this case it provides a more compact and maintainable model compared with the state-based approach, where the number of executable transitions has been reduced from 9 to 3.

The Retransmission Page. During connection set up and close down retransmissions occur when DCCP has not received a response from its peer within a

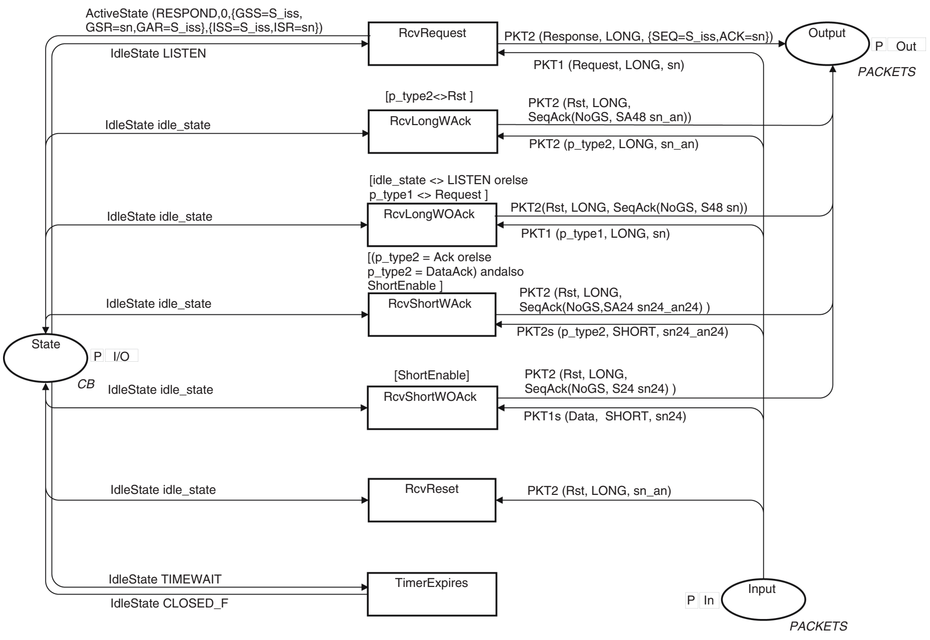

ActiveState (RESPOND,0,{GSS=S_iss, GSR=sn,GAR=S_iss},{ISS=S_iss,ISR=sn})
IdleState LISTEN
RcvRequest
PKT2 (Response, LONG, {SEQ=S_iss,ACK=sn})
PKT1 (Request, LONG, sn)
Output
P Out
PACKETS
[p_type2<>Rst]
IdleState idle_state
RcvLongWAck
PKT2 (Rst, LONG, SeqAck(NoGS, SA48 sn_an))
PKT2 (p_type2, LONG, sn_an)
[idle_state <> LISTEN orelse p_type1 <> Request]
RcvLongWOAck
PKT2(Rst, LONG, SeqAck(NoGS, S48 sn))
PKT1 (p_type1, LONG, sn)
[(p_type2 = Ack orelse p_type2 = DataAck) andalso ShortEnable]
RcvShortWAck
PKT2 (Rst, LONG, SeqAck(NoGS,SA24 sn24_an24))
PKT2s (p_type2, SHORT, sn24_an24)
State
P I/O
CB
[ShortEnable]
RcvShortWOAck
PKT2 (Rst, LONG, SeqAck(NoGS, S24 sn24))
PKT1s (Data, SHORT, sn24)
RcvReset
PKT2 (Rst, LONG, sn_an)
IdleState TIMEWAIT
IdleState CLOSED_F
TimerExpires
Input
P In
PACKETS

Fig. 17. The IdleState page

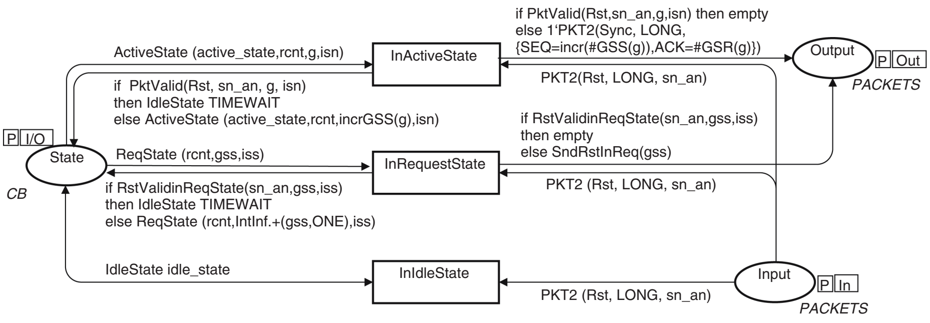

if PktValid(Rst,sn_an,g,isn) then empty else 1‘PKT2(Sync, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
ActiveState (active_state,rcnt,g,isn)
InActiveState
Output
P Out
PACKETS
if PktValid(Rst, sn_an, g, isn) then IdleState TIMEWAIT else ActiveState (active_state,rcnt,incrGSS(g),isn)
PKT2(Rst, LONG, sn_an)
P I/O
State
CB
if RstValidinReqState(sn_an,gss,iss) then empty else SndRstInReq(gss)
ReqState (rcnt,gss,iss)
InRequestState
PKT2 (Rst, LONG, sn_an)
if RstValidinReqState(sn_an,gss,iss) then IdleState TIMEWAIT else ReqState (rcnt,IntInf.+(gss,ONE),iss)
IdleState idle_state
InIdleState
PKT2 (Rst, LONG, sn_an)
Input
P In
PACKETS

Fig. 18. The RcvReset page

specified period (a timer expires). We group all transitions related to retransmission into a CPN page called the Retransmission page shown in Fig. 19. Retransmission in each state is modelled by the transition corresponding to the state's name: Retrans_REQUEST; RetransShort_PARTOPEN; RetransLong_PARTOPEN; Retrans_CLOSEREQ and Retrans_CLOSING. In PARTOPEN the client can retransmit either an Ack or a DataAck with short or long sequence numbers. Each retransmission increases the retransmission counter (rcnt) by one. This approach clearly shows the states in which retransmission is possible. For example, retransmission does not occur in the RESPOND state.

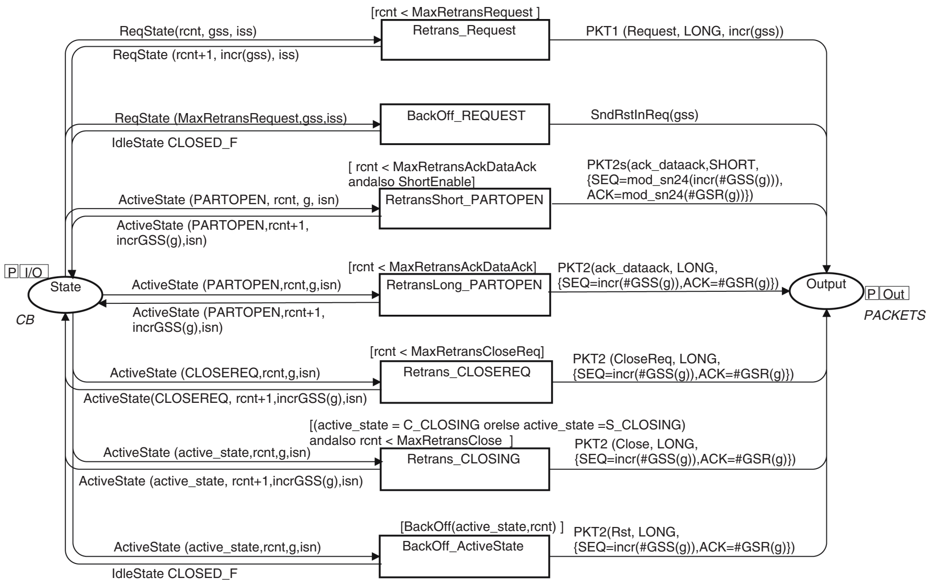

[rcnt < MaxRetransRequest]
Retrans_Request
ReqState(rcnt, gss, iss)
ReqState (rcnt+1, incr(gss), iss)
PKT1 (Request, LONG, incr(gss))
BackOff_REQUEST
ReqState (MaxRetransRequest,gss,iss)
IdleState CLOSED_F
SndRstInReq(gss)
[rcnt < MaxRetransAckDataAck andalso ShortEnable]
RetransShort_PARTOPEN
ActiveState (PARTOPEN, rcnt, g, isn)
ActiveState (PARTOPEN,rcnt+1, incrGSS(g),isn)
PKT2s(ack_dataack,SHORT, {SEQ=mod_sn24(incr(#GSS(g))), ACK=mod_sn24(#GSR(g))})
[rcnt < MaxRetransAckDataAck]
RetransLong_PARTOPEN
ActiveState (PARTOPEN,rcnt,g,isn)
ActiveState (PARTOPEN,rcnt+1, incrGSS(g),isn)
PKT2(ack_dataack, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
P I/O
State
CB
Output
P Out
PACKETS
[rcnt < MaxRetransCloseReq]
Retrans_CLOSEREQ
ActiveState (CLOSEREQ,rcnt,g,isn)
ActiveState(CLOSEREQ, rcnt+1,incrGSS(g),isn)
PKT2 (CloseReq, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
[(active_state = C_CLOSING orelse active_state =S_CLOSING) andalso rcnt < MaxRetransClose]
Retrans_CLOSING
ActiveState (active_state,rcnt,g,isn)
ActiveState (active_state, rcnt+1,incrGSS(g),isn)
PKT2 (Close, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
[BackOff(active_state,rcnt)]
BackOff_ActiveState
ActiveState (active_state,rcnt,g,isn)
IdleState CLOSED_F
PKT2(Rst, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})

Fig. 19. The Retransmission page

When the counter reaches the maximum retransmission value, DCCP resets the connection and enters the CLOSED state. These actions are modelled by the transition BackOff_REQUEST for the REQUEST state and BackOff_ActiveState for the RESPOND, PARTOPEN, CLOSEREQ and CLOSING states. The guard function, BackOff, checks whether the number of retransmissions has reached the maximum value or not. Although no Response packet is retransmitted in RESPOND, DCCP enters the CLOSED state after holding the RESPOND state for longer than the backoff timer period (4MPL). The common backoff timer procedure for active states is captured in the one transition (BackOff_ActiveState), avoiding repetition of this behaviour in each state page.

4 Conclusion and Future Work

This paper has presented our approach to the development of a formal model of the connection management and synchronization procedures of the Internet's Datagram Congestion Control Protocol. We discuss how our CPN model was refined using an incremental approach while the specification was being developed by IETF. We started with a small model of an early Internet Draft and illustrated how to use hierarchical CPNs to structure the model. This model used a state-based approach. We then showed how the model's hierarchical structure and data structures evolved to capture DCCP's specification faithfully as it was progressively developed by IETF. At various stages of the development we analysed the model and found errors in the specification. Serious errors, such as chatter, were reported to IETF and their solutions verified before DCCP became a standard (RFC 4340).

We found that as the state-based CPN model grew, a lot redundancy crept into the model. This was especially the case when including the synchronization procedures. The redundancy occurs when the receipt of the same packet in different states requires the same processing. We revised parts of the model using an event processing style so that similar events occurring in various states are regrouped into one transition. The final model is significantly more compact than the previous models.

Significant effort was put into validating the model. This can be facilitated by firstly using the state-based approach so that it is easy to check that an action is defined for the receipt of each packet in each state. The state-based approach is easier to read, however, the judicious introduction of the event processing style in significant parts of the model made it easier to maintain, as our interaction with IETF proved.

The overall benefits of the approach include: the model is formal and hence removes ambiguity; the model gathers together disparate parts of the RFC that are specified by a combination of narrative, a state machine and pseudo code in different sections; the concurrency inherent in the protocol is retained; and the CPN model can be analysed to obtain results which are fed back into the standardisation process to improve the specification before it becomes a standard.

Our incremental approach to the analysis of the model requires consideration of four different channel types: ordered without loss; ordered with loss; reordered without loss; and reordered with loss. To facilitate maintenance of the model, we are currently integrating these different channel types into a single CPN model. We are also interested in modelling and analysing the feature negotiation procedures.

Acknowledgements

The authors gratefully acknowledge the constructive comments of the anonymous referees which have helped to improve this paper.

References

1. Billington, J., Gallasch, G.E., Han, B.: A Coloured Petri Net Approach to Protocol Verification. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets, Advances in Petri Nets. LNCS, vol. 3098, pp. 210–290. Springer, Heidelberg (2004)
2. Floyd, S., Handley, M., Kohler, E.: Problem Statement for the Datagram Congestion Control Protocol (DCCP), RFC 4336 (March 2006) Available via http://www.rfc-editor.org/rfc/rfc4336.txt
3. Floyd, S., Kohler, E.: Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control, RFC 4341 (March 2006) Available via http://www.rfc-editor.org/rfc/rfc4341.txt

4. Floyd, S., Kohler, E., Padhye, J.: Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC), RFC 4342 (March 2006) Available via http://www.rfc-editor.org/rfc/rfc4342.txt
5. Gallasch, G.E., Billington, J., Vanit-Anunchai, S., Kristensen, L.M.: Checking Safety Properties On-The-Fly with the Sweep-line Method. In: International Journal on Software Tools for Technology Transfer, Springer, Heidelberg (to appear 2007)
6. Han, B.: Formal Specification of the TCP Service and Verification of TCP Connection Management. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, South Australia (December 2004)
7. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. vol. 1, Basic Concepts. Monographs in Theoretical Computer Science. Springer, Heidelberg (2nd edition, 1997)
8. Kohler, E., Handley, M., Floyd, S.: Designing DCCP: Congestion Control Without Reliability. In: Proceedings of the 2006 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'06), Pisa, Italy, pp. 27–38 (September 11-15, 2006)
9. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol, draft-ietf-dccp-spec-6 (February 2004) Available via http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-spec-06.txt
10. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol, draft-ietf-dccp-spec-7 (July 2004) Available via http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-spec-07.txt
11. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol, draft-ietf-dccp-spec-11 (March 2005) Available via http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-spec-11.txt
12. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol, draft-ietf-dccp-spec-13 (December 2005) Available via http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-spec-13.txt
13. Kohler, E., Handley, M., Floyd, S.: SUBSTANTIVE DIFFERENCES BETWEEN draft-ietf-dccp-spec-11 AND draft-ietf-dccp-spec-12 (December 2005) Available via http://www.read.cs.ucla.edu/dccp/diff-spec-11-12-explain.txt
14. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol, RFC 4340 (March 2006) Available via http://www.rfc-editor.org/rfc/rfc4340.txt
15. Kohler, E., Handley, M., Floyd, S.: SUBSTANTIVE DIFFERENCES BETWEEN draft-ietf-dccp-spec-13 AND RFC 4340 March (2006) Available via http://www.read.cs.ucla.edu/dccp/diff-spec-13-rfc-explain.txt
16. Kohler, E., Handley, M., Floyd, S., Padhye, J.: Datagram Congestion Control Protocol, draft-ietf-dccp-spec-5 (October 2003) Available via http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-spec-05.txt
17. Kongprakaiwoot, T.: Verification of the Datagram Congestion Control Protocol using Coloured Petri Nets. Master's thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, South Australia (November 2004)
18. University of Aarhus. Design/CPN Online. Department of Computer Science (2004) Available via http://www.daimi.au.dk/designCPN/
19. Vanit-Anunchai, S., Billington, J.: Initial Result of a Formal Analysis of DCCP Connection Management. In: Proceedings of Fourth International Network Conference (INC 2004), pp. 63–70, Plymouth, UK, 6-9 July 2004. University of Plymouth (2004)

20. Vanit-Anunchai, S., Billington, J.: Effect of Sequence Number Wrap on DCCP Connection Establishment. In: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Monterey, California, USA, 11-13 September 2006, pp. 345–354. IEEE Computer Society Press, Washington (2006)
21. Vanit-Anunchai, S., Billington, J.: Chattering Behaviour in the Datagram Congestion Control Protocol. IEE Electronics Letters 41(21), 1198–1199 (2005)
22. Vanit-Anunchai, S., Billington, J., Gallasch, G.E.: Sweep-line Analysis of DCCP Connection Management. In: Proceeding of the Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Technical Report, DAIMI PB-579, Aarhus, Denmark, 24-26 October, pp. 157–175, Department of Computer Science, University of Aarhus. (2006) Available via http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/
23. Vanit-Anunchai, S., Billington, J., Kongprakaiwoot, T.: Discovering Chatter and Incompleteness in the Datagram Congestion Control Protocol. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 143–158. Springer, Heidelberg (2005)

The ComBack Method – Extending Hash Compaction with Backtracking

Michael Westergaard, Lars Michael Kristensen*, Gerth Stølting Brodal, and Lars Arge

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
{mw,kris,gerth,large}@daimi.au.dk

Abstract. This paper presents the ComBack method for explicit state space exploration. The ComBack method extends the well-known hash compaction method such that full coverage of the state space is guaranteed. Each encountered state is mapped into a compressed state descriptor (hash value) as in hash compaction. The method additionally stores for each state an integer representing the identity of the state and a backedge to a predecessor state. This allows hash collisions to be resolved on-the-fly during state space exploration using backtracking to reconstruct the full state descriptors when required for comparison with newly encountered states. A prototype implementation of the ComBack method is used to evaluate the method on several example systems and compare its performance to related methods. The results show a reduction in memory usage at an acceptable cost in exploration time.

1 Introduction

Explicit state space exploration is one of the main approaches to verification of finite-state concurrent systems. The underlying idea is to enumerate all reachable states of the system under consideration, and it has been implemented in computer tools such as SPIN [9], Murϕ [10], CPN Tools [18], and LoLa [20].

The main drawback of verification methods based on state space exploration is the *state explosion problem* [25], and several reduction methods have been developed to alleviate this inherent complexity problem. For explicit state space exploration these include: methods that explore only a subset of the state space directed by the verification question [17,24]; methods that delete states from memory during state space exploration [6,3,1]; methods that store states in a compact manner in memory [8,12,5]; and methods that use external storage to store the set of visited states [22]. Another approach is symbolic model checking using, e.g., binary decision diagrams [2] or multi-valued decision diagrams [13].

Of particular interest in the context of this paper is the *hash compaction method* [27,21], a method to reduce the amount of memory used to store states.

* Supported by the Carlsberg Foundation and the Danish Research Council for Technology and Production.

Hash compaction uses a hash function H to map each encountered state s into a fixed-sized bit-vector $H(s)$ called the *compressed state descriptor* which is stored in memory as a representation of the state. The *full state descriptor* is not stored in memory. Thus, each discovered state is represented compactly using typically 32 or 64 bits. The disadvantage of hash compaction is that two different states may be mapped to the same compressed state descriptor which implies that the hash compaction method may not explore all reachable states. The probability of *hash collisions* can be reduced by using multiple hash functions [21], but the method still cannot guarantee full coverage of the state space. If the intent of state space exploration is to find (some) errors, this is acceptable. If, however, the goal is to prove the absence of errors, discarding parts of the state space is not acceptable, meaning that hash compaction is mainly suited for error detection.

The idea of the ComBack method is to augment the hash compaction method such that hash collisions can be resolved during state space exploration. This is achieved by assigning a unique *state number* to each visited state and by storing, for each compressed state descriptor, a list of state numbers that have been mapped to this compressed state descriptor. This information is stored in a *state table*. Furthermore, a *backedge table* stores a *backedge* for each visited state. A backedge for a state s consists of a transition t and a state number n, such that executing transition t in the predecessor state s' with state number n leads to s. The backedges stored in the backedge table determine a spanning tree rooted in the initial state for the partial state space currently explored. The backedge table makes it possible, given the state number of a visited state s, to *backtrack* to the initial state and thereby obtain a sequence of transitions (corresponding to a path in the state space) which, when executed from the initial state, leads to s, which makes it possible to reconstruct the full state descriptor of s.

A potential hash collision is detected whenever a newly generated state s is mapped to a compressed state descriptor $H(s)$ already stored in the state table. From the compressed state descriptor and the state table we obtain the list of visited state numbers mapped to this compressed state descriptor. Using the backedge table, the full state descriptor can be reconstructed for each of these states and compared to the newly generated state s. If none of the full state descriptors for the already stored state numbers is equal to the full state descriptor of s, then s has not been visited before, and a hash collision has been detected. The state s is therefore assigned a new state number which is appended to the list of state numbers for the given compressed state descriptor, and a backedge for s is inserted into the backedge table. Otherwise, s was identical to an already visited state and no action is required.

The rest of this paper is organised as follows. Section 2 introduces the basic notation and presents the hash compaction algorithm. Section 3 introduces the ComBack method using a small example, and Sect. 4 formally specifies the ComBack algorithm. Section 5 presents several variants of the basic ComBack algorithm, and Sect. 6 presents a prototype implementation together with experimental results obtained on a number of example systems. Finally, in Sect. 7, we sum up the conclusions and discuss future work. The reader is assumed to be familiar with the basic ideas of explicit state space exploration.

2 Background

The ComBack method has been developed in the context of Coloured Petri nets (CP-nets or CPNs) [11], but applies to many other modelling languages for concurrent systems such as PT-nets [19], CCS [16], and CSP [7]. We therefore formulate the ComBack method in the context of (finite) labelled transition systems to make the presentation independent of a concrete modelling language.

Definition 1 (Labelled Transition System). *A labelled transition system (LTS) is a tuple $\mathcal{S} = (S, T, \Delta, s_I)$, where S is a finite set of* ***states****, T is a finite set of* ***transitions****, $\Delta \subseteq S \times T \times S$ is the* ***transition relation****, and $s_I \in S$ is the* ***initial state****.*

In the rest of this paper we assume that we are given a labelled transition system $\mathcal{S} = (S, T, \Delta, s_I)$. Let $s, s' \in S$ be two states and $t \in T$ a transition. If $(s, t, s') \in \Delta$, then t is said to be *enabled* in s and the *occurrence* (execution) of t in s leads to the state s'. This is also written $s \xrightarrow{t} s'$. An *occurrence sequence* is an alternating sequence of states s_i and transitions t_i written $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$ and satisfying $s_i \xrightarrow{t_i} s_{i+1}$ for $1 \leq i \leq n-1$. For the presentation of the ComBack method, we initially assume that transitions are *deterministic*, i.e., if $s \xrightarrow{t} s'$ and $s \xrightarrow{t} s''$ then $s' = s''$. This holds for transitions in, e.g., PT-nets and CP-nets. In Sect. 5 we show how to extend the ComBack method to modelling languages with non-deterministic transitions.

We use $\rightarrow^*$ to denote the transitive and reflexive closure of Δ, i.e., $s \rightarrow^* s'$ if and only if there exists an occurrence sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$, $n \geq 1$, with $s = s_1$ and $s' = s_n$. A state s' is *reachable* from s if and only if $s \rightarrow^* s'$, and $\mathsf{reach}(s) = \{\, s' \in S \mid s \rightarrow^* s' \,\}$ denotes the set of states reachable from s. The *state space* of a system is the directed graph (V, E) where $V = \mathsf{reach}(s_I)$ is the set of nodes and $E = \{(s, t, s') \in \Delta \mid s, s' \in V\}$ is the set of edges.

The standard algorithm for explicit state space exploration relies on two data structures: a *state table* storing the states that have been discovered until now, and a *waiting set* containing the states for which successor states have not yet been calculated. The state table can be implemented as a hash table, and the waiting set can be implemented, e.g., as a stack or a fifo-queue if depth-first or breadth-first exploration is desired. The state table and the waiting set are initialised to contain the initial state and the algorithm terminates when the waiting set is empty, at which point the state table contains the reachable states.

The basic idea of the *hash compaction method* [27,21] is to use a *hash function* H mapping from states S into the set of bit-strings of some fixed length. Instead of storing the *full state descriptor* in the state table for each visited state s, only the *compressed state descriptor* (hash value) $H(s)$ is stored. The waiting set still stores full state descriptors. Algorithm 1 gives the basic hash compaction

Algorithm 1. Basic Hash Compaction Algorithm

```
1: StateTable.Init(); StateTable.Insert($H(s_I)$)
2: WaitingSet.Init(); WaitingSet.Insert($s_I$)
3:
4: while $\neg$ WaitingSet.Empty() do
5:    $s \leftarrow$ WaitingSet.Select()
6:    for all $t, s'$ such that $(s, t, s') \in \Delta$ do
7:       if $\neg$ StateTable.Contains($H(s')$) then
8:          StateTable.Insert($H(s')$)
9:          WaitingSet.Insert($s'$)
```

algorithm [27]. The state table and the waiting set are initialised in lines 1–2 with the compressed and full state descriptors for the initial state s_I, respectively. The algorithm then executes a while-loop (lines 4-9) until the waiting set is empty. In each iteration of the while loop, a state s is selected and removed from the waiting set (line 5) and each of the successor states s' of s are calculated and examined (lines 6-9). If the compressed state descriptor $H(s')$ for s' is not in the state table, then s' has not been visited before, and $H(s')$ is added to the state table and s' is added to the waiting set. If the compressed state descriptor $H(s')$ for s' is already in the state table, the assumption of the hash compaction method is that s' has already been visited. The advantage of the hash compaction method is that the number of bytes stored per state is heavily reduced compared to storing the full state descriptor, which can be several hundreds of bytes for complex systems. The disadvantage is that the method cannot guarantee full coverage of the state space.

Figure 1 shows an example state space which will also be used when introducing the ComBack method in the next section. Figure 1(left) shows the full state space consisting of the states $s_1, s_2, \ldots, s_6$. The initial state is s_1. The compressed state descriptors h_1, h_2, h_3, h_4 have been written to the upper right of each state. As an example, it can be seen that the states s_3, s_5, and s_6 are mapped to the same compressed state descriptor h_3. Figure 1(right) shows the part of the state space explored by the hash compaction method. The hash compaction method will consider the states s_3, s_5, and s_6 to be the same state since they are mapped to the same compressed state descriptor h_3. As a result, the hash compaction method does not explore the full state space.

Several improvements have been developed for the basic hash compaction method to reduce the probability of not exploring the full state space [21]. None of these improvements guarantee full coverage of the state space. For the purpose of this paper it therefore suffices to consider the basic hash compaction algorithm.

3 The ComBack Method

The basic idea of the ComBack method is similar to that of the hash compaction method: instead of storing the full state descriptors, a hash function is used to calculate a compressed state descriptor. When using hash compaction, the main problem is *hash collisions*, i.e., that states with different full state descriptors (such as s_3, s_5, and s_6 in Fig. 1) are mapped to the same compressed state descriptor. The ComBack method addresses this problem by comparing the full state descriptors whenever a new state is generated for which the compressed state descriptor is already stored in the state table. This is, however, done without storing the full state descriptors for the states in the state table. Instead the full state descriptors of states in the state table are reconstructed on-demand using *backtracking* to resolve hash collisions. The reconstruction of full state descriptors using backtracking is achieved by augmenting the hash compaction algorithm in the following ways:

1. A *state number* $N(s)$ (integer) is assigned to each visited state s.
2. The state table stores for each compressed state descriptor a *collision list* of state numbers for visited states mapped to this compressed state descriptor.
3. A *backedge table* is maintained which for each state number $N(s)$ of a visited state s stores a *backedge* consisting of a transition t and a state number $N(s')$ of a visited state s' such that $s' \xrightarrow{t} s$.

The augmented state table makes it possible, given a compressed state descriptor $H(s)$ for a newly generated state s, to obtain the state numbers for the visited states mapped to the compressed state descriptor $H(s)$. For each such state number $N(s')$ of a state s', the backedge table can be used to obtain the sequence of transitions, $t_1 t_2 \cdots t_n$, on some path (occurrence sequence) in the state space leading from the initial state s_I to s'. As we have initially assumed that transitions are deterministic, executing this occurrence sequence starting in the initial state will reconstruct the full state descriptor for s'. It is therefore possible to compare the full state descriptor of the newly generated state s to the full state descriptor of s' and thereby determine whether s has already been encountered.

Figure 2 (left) shows a snapshot of state space exploration using the ComBack method on the example that was introduced in Fig. 1. The snapshot represents the situation where the successors of the initial state s_1 have been generated, and the states s_2 and s_6 are the states currently in the waiting set. The state number assigned to each state is written inside a box to the upper left of each

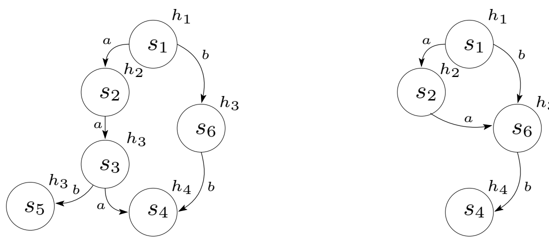
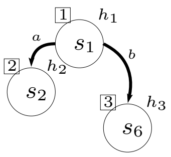


Fig. 1. Full state space (left) and state space explored using hash compaction (right)

state. Figure 2 (middle) shows the contents of the state table, which for each compressed state descriptor h_i lists the state numbers mapped to h_i. Figure 2 (right) shows the contents of the backedge table. The backedge table gives for each state number $N(s)$ a pair $(N(s'), t)$, consisting of the state number $N(s')$ of a predecessor state s' and a transition t such that $s' \xrightarrow{t} s$. As an example, for state number 3 (which is state s_6) the backedge table specifies the pair $(1, b)$ corresponding to the edge in the state space going from state s_1 to state s_6 labelled with the transition b. For the initial state, which by convention always has state number 1, no backedge is specified since backtracking will always be stopped at the initial state.

Assume that s_2 is the next state removed from the waiting set. It has a single successor state s_3 which is mapped to the compressed state descriptor h_3 (see Fig. 1). A lookup in the state table shows that for the compressed state descriptor h_3 we already have a state with state number 3 stored. We therefore need to reconstruct the full state descriptor for state number 3 in order to determine whether s_3 is a newly discovered state. The reconstruction is done in two phases. The first phase uses the backedge table to obtain a sequence of transitions which, when executed from the initial state, leads to the state with number 3. A lookup in the backedge table for the state with state number 3 yields the pair $(1, b)$. Since 1 represents the initial state, the backtracking terminates with the transition sequence consisting of b. In the second phase, we use the transition relation $\varDelta$ for the system to execute the transition b in the initial state and obtain the full state descriptor for state number 3 (which is s_6). We can now compare the full state descriptors s_3 and s_6. Since these are different, s_3 is a new state and assigned state number 4, which is added to the state table by appending it to the collision list for the compressed state descriptor h_3. In addition s_3 is added to the waiting set, and an entry $(2, a)$ is added to the backedge table for state number 4 in case we will have to reconstruct s_3 later. Figure 3 shows the state space explored, the state table, and the backedge table after processing s_2.

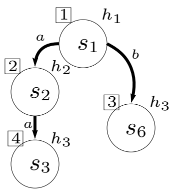
The waiting set now contains s_3 and s_6. Assume that s_3 is selected from the waiting set. The two successor states s_4 and s_5 will be generated. First, we will check whether s_4 has already been generated. As s_4 has the compressed state descriptor h_4, which has no state numbers in its collision list, it is new, and it is assigned state number 5, and an entry $(4, a)$ is added to the backedge table. Then we check if s_5 is new. State s_5 has the compressed state descriptor h_3 and a lookup in the state table yields the collision list consisting of states number



h_1	1
h_2	2
h_3	3

1	
2	$(1, a)$
3	$(1, b)$

Fig. 2. Before s_2 is processed: state space explored (left), state table (middle), and backedge table (right)



h_1	1	
h_2	2	
h_3	3	4

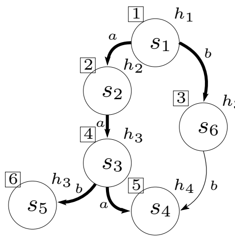
1	
2	$(1, a)$
3	$(1, b)$
4	$(2, a)$

Fig. 3. After processing s_2: state space explored (left), state table (middle), and backedge table (right)

3 and 4. Using the backedge table, we obtain the two corresponding transition sequences: $(1, b)$ and $(2, a)(1, a)$. Executing the occurrence sequences: $s_1 \xrightarrow{b} s_6$ and $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3$ yields the full state descriptors for s_3 and s_6. By comparison with the full state descriptor for s_5 it is concluded that s_5 is new and the state table, the waiting set, and the backedge table are updated accordingly.

When state s_3 has been processed, the waiting set contains the states s_4, s_5, and s_6. The processing of s_4 and s_5 does not result in any new states as these two states do not have successor states. Consider the processing of s_6. We will tentatively denote the full state descriptor for the successor of s_6 corresponding to s_4 by s' as the algorithm has not yet determined that it is equal to s_4. State s' has the compressed state descriptor h_4 and a lookup in the state table shows that we have a single state with number 5 stored for h_4. The backedge table is then used starting from state number 5 to obtain the backedges $(4, a)$, $(2, a)$, and $(1, a)$. Executing the corresponding occurrence sequence $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} s_4$ yields full state descriptor for s_4, and we conclude that this full state descriptor is equal to s', so s' has already been visited and no changes are required to the state table, the waiting set or the backedge table.

Figure 4 shows the situation after state s_6 has been processed. The thick edges correspond to the backedges stored in the backedge table. It can be seen that the backedges stored in the backedge table determine a spanning tree rooted in the initial state in all stages of the construction (Figs. 2–4).



h_1	1		
h_2	2		
h_3	3	4	6
h_4	5		

1	
2	$(1, a)$
3	$(1, b)$
4	$(2, a)$
5	$(4, a)$
6	$(4, b)$

Fig. 4. After processing s_6: state space explored (left), state table (middle), and backedge table (right)

4 The ComBack Algorithm

The ComBack algorithm introduced in the previous section is listed in Algorithm 2. The first part of the algorithm (lines 1–4) initialises the global data structures. The global variable m is used to enumerate the states, i.e., assign state numbers to states, and is initially 1 since the initial state is the first state considered. The state table has an Insert operation which takes a compressed state descriptor and a state number and appends the state number to the collision list for the compressed state descriptor. The waiting set stores pairs consisting of a full state descriptor and its number. The state number is needed when creating the backedge for a newly discovered state. The backedge table stores pairs consisting of a state number and a transition label. The empty backedge denoted $\perp$ is initially inserted in the backedge table for state number 1 (the initial state).

The algorithm then executes a while-loop (lines 6–13) until the waiting set is empty. In each iteration of the while-loop, a pair, (s, n'), consisting of a state

Algorithm 2. The ComBack Algorithm

```
 1: $m \leftarrow 1$
 2: StateTable.Init(); StateTable.Insert($H(s_I), 1$))
 3: WaitingSet.Init(); WaitingSet.Insert($s_I, 1$)
 4: BackEdgeTable.Init(); BackEdgeTable.Insert($1, \perp$)
 5:
 6: while $\neg$ WaitingSet.Empty() do
 7:   $(s, n')$ $\leftarrow$ WaitingSet.Select()
 8:   for all $t, s'$ such that $(s, t, s') \in \Delta$ do
 9:     if $\neg$ Contains($s'$) then
10:       $m \leftarrow m + 1$
11:       StateTable.Insert($H(s'), m$)
12:       WaitingSet.Insert($s', m$)
13:       BackEdgeTable.Insert($m, (n', t)$)
14:
15: proc Contains($s'$) is
16:   for all $n \in$ StateTable.Lookup($H(s')$) do
17:     if Matches($n, s'$) then
18:       return tt
19:   return ff
20:
21: proc Matches($n, s'$) is
22:   return $s'$ = Reconstruct($n$)
23:
24: proc Reconstruct($n$) is
25:   if $n = 1$ then
26:     return $s_I$
27:   else
28:     $(n', t)$ $\leftarrow$ BackEdgeTable.Lookup($n$)
29:     $s$ $\leftarrow$ Reconstruct($n'$)
30:     return Execute($s, t$)
```

and its state number is selected from the waiting set (line 7) and each of the successor states, s', of s is examined (lines 8–13). Whether a successor state, s', is a newly discovered state is determined using the CONTAINS procedure, which will be explained below. If s' is a newly discovered state, m is incremented by one to obtain the state number assigned to s', the state number for s' is appended to the collision list associated with the compressed state descriptor $H(s')$, and (n', t) is inserted as a backedge in the backedge table for the state s' which has been given state number m.

The procedure CONTAINS (lines 15–19) is used to determine whether a newly generated state s' has been visited before. The procedure looks up the collision list for the compressed state descriptor $H(s')$ for s', and for each state number, n, in the collision list it checks if s' corresponds to n using the MATCHES procedure. If a reconstructed state descriptor is identical to s', then s' has already been visited and tt (true) is returned. Otherwise ff (false) is returned. The procedure MATCHES (lines 21–22) reconstructs the full state descriptor corresponding to n using RECONSTRUCT procedure and returns whether it is equal to s'.

The procedure RECONSTRUCT recursively backtracks using the backedge table to reconstruct the full state descriptor for state number n. The function recursively finds the state number of a predecessor using the backedge table and calculates the full state descriptor using the EXECUTE procedure. The procedure exploits the convention that the initial state has number 1 to determine when to stop the recursion. The EXECUTE procedure (not shown) uses the transition relation Δ to compute the state resulting from an occurrence of the transition t in the state s, i.e., if $(s, t, s') \in \Delta$ then EXECUTE$(s, t) = s'$. This is well-defined since we have assumed that transitions are deterministic.

It can be seen that the ComBack algorithm is very similar to the standard algorithm for state space exploration. The main difference is that determining whether a state has already been visited relies on the CONTAINS procedure which uses the backedge table to reconstruct the full state descriptors before the comparison with a newly generated state is done. Since the backedge table at any time during state exploration determines a spanning tree rooted in the initial state for the currently explored part of the state space, we can reconstruct the full state descriptor for any visited state. It follows that the ComBack algorithm terminates after having explored all reachable states exactly once.

Space Usage. The ComBack algorithm explores the full state space at the expense of using more memory per state than hash compaction and by using time on reconstruction of full state descriptors. We will now discuss these two issues in more detail. First we consider memory usage. Let w_N denote the number of bits used to represent a state number, and let w_H denote the number of bits in a compressed state descriptor. Let $|h_i|$ denote the number of reachable states mapped to the compressed state descriptor h_i. The entry corresponding to h_i in the state table can be stored as a pair consisting of the compressed state descriptor and a counter of size w_c specifying the length of an array of state numbers (the collision list). The total amount of memory used to store the states whose compressed state descriptor is h_i is therefore given by $w_H + w_c + |h_i| \cdot w_N$. Considering all compressed state descriptors, the worst-case memory usage occurs if all collision lists have length 1. This means that the worst-case memory usage for the state table is:

$$|\mathsf{reach}(s_I)| \cdot (w_H + w_c + w_N)$$

We need at least $w_N = \lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits for storing unique numbers for each state and $w_c = \lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits for storing the number of states in each collision list. The worst-case memory usage for the elements in the state table is therefore:

$$|\mathsf{reach}(s_I)| \cdot (w_H + 2 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil)$$

Consider now the backedge table. The entries can be implemented as an array where entry i specifies the backedge associated with state number i. If we enumerate all transitions, each transition in a backedge can be represented using $\lceil \log_2 |T| \rceil$ bits. Each state number in a backedge can be represented using $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits. Observing that each reachable state will have one entry in the backedge table upon termination this implies that the memory used for the elements in the backedge table is given by:

$$|\mathsf{reach}(s_I)| \cdot (\lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil)$$

The above means that the total amount memory used for the elements in the state table and the backedge table is in worst-case given by:

$$|\mathsf{reach}(s_I)| \cdot (w_H + 3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil)$$

This is $3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil$ bits more per visited state than the hash compaction method. The ComBack method and the hash compaction method both store the full state descriptor for those states that are in the waiting set, but the ComBack method additionally stores the state number of each state in the waiting set which implies that the ComBack method uses $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ more bits per state in the waiting set. In reality, we will not know $|\mathsf{reach}(s_I)|$ in advance, and we will therefore use a machine word (w bits) for storing state numbers. If we furthermore assume that we store each transition using a machine word and use a hash function generating compressed state descriptors of size $w_H = w$, we use a total of $5 \cdot w$ bits or 5 machine words per state, corresponding to 20 bytes on a 32-bit architecture.

Time Analysis. Let us now consider the additional time used by the ComBack algorithm for reconstruction of full state descriptors. Let $\hat{h}_i = \{s_1, s_2, \ldots, s_n\}$ denote the states that are mapped to given compressed state descriptor h_i and assume that they are discovered in this order. The first state s_1 mapped to h_i will not result in a state reconstruction, but when state s_j is discovered the first time it will cause a reconstruction of the states $s_1, s_2, \ldots s_{j-1}$. This means that the number of reconstructions caused by the first discovery of each of the states is given by:

$$\textstyle\sum_{j=1}^{|\hat{h}_i|} (j-1) = \frac{|\hat{h}_i| \cdot (|\hat{h}_i| - 1)}{2}$$

Any additional input edge of an already discovered state mapped to h_i will in worst-case cause all other discovered states to be regenerated. In the worst case, the additional input edges are discovered after all $|\hat{h}_i|$ states have been discovered for the first time. Let $\mathsf{in(s)}$ denote the number of input edges for a state s. The number of reconstructions caused by additional input edges is then given by:

$$|\hat{h_i}| \cdot \sum_{s_j \in \hat{h}_i} (\mathsf{in}(s_j) - 1)$$

This means that the total number of state reconstructions for a given compressed state descriptor h_i is given by:

$$\begin{aligned}\tfrac{|\hat{h}_i|\cdot(|\hat{h}_i|-1)}{2} + |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} (\mathsf{in}(s_j) - 1) &= \tfrac{1}{2}|\hat{h}_i|^2 - \tfrac{|\hat{h}_i|}{2} + |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}(s_j) - |\hat{h}_i|^2 \\ &= -\tfrac{1}{2}|\hat{h}_i|^2 - \tfrac{|\hat{h}_i|}{2} + |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \\ &\leq |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j)\end{aligned}$$

Let $\hat{H} = \{\, H(s) \mid s \in \mathsf{reach}(s_I) \,\}$ denote the set of compressed state descriptors for the set of reachable states. The number of reconstructions used for the entire state space exploration can be then be approximated by:

$$\begin{aligned}\textstyle\sum_{h_i \in \hat{H}} |\hat{h}_i| \cdot \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) &\leq \textstyle\sum_{h_i \in \hat{H}} \left(\max_{h_k \in \hat{H}} |h_k| \cdot \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j)\right) \\ &= \textstyle\max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{h_i \in \hat{H}} \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \\ &= \textstyle\max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{s \in \mathsf{reach}(s_I)} \mathsf{in}\,(s_j)\end{aligned}$$

If we assume that we are using a good hash function for computing the compressed state descriptors, then $|\hat{h}_i|$ will in practice be small (at most 2 or 3). This means that the total number of state reconstructions will be close to the sum of the in-degrees of all reachable states which is equal to number of edges in the full state space. A poor hash function will cause many state reconstructions which in turn will seriously affect the run-time performance of the algorithm. In Sect. 6 we will show how to obtain a good hash function in the context of CP-nets. If the backedge table is implemented as an array, we get a constant look-up time, and a state can be reconstructed in time proportional to the length of the path.

The above is summarised in the following theorem where $\{0,1\}^{w_H}$ denotes the set of bit strings of length w_H.

Theorem 1. *Let $\mathcal{S} = (S, T, \Delta, s_I)$ be a labelled transition system and $H : S \to \{0,1\}^{w_H}$ be a hash function. The ComBack algorithm in Algorithm 2 terminates after having explored all reachable states of $\mathcal{S}$ exactly once. The elements in the state table and the backedge table can be represented using:*

$$|\mathsf{reach}(s_I)| \cdot (w_H + 3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil) \textit{ bits}$$

The total number of state reconstructions during exploration is bounded by:

$$\max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{s \in \mathsf{reach}(s_I)} \mathit{in}\,(s_j)$$

5 Variants and Extensions

In this section, we sketch several variants of the basic ComBack algorithm. Variants 1 and 2 are aimed at reducing time usage while Variants 3 and 4 are aimed at reducing memory usage. Variant 5 shows how the ComBack method can be used for modelling languages with non-deterministic transitions.

Variant 1: Path Optimisation. The amount of time used on reconstruction of a state s is proportional to the length of the occurrence sequence leading to s stored in the backedge table. If the state space is constructed in a breadth-first order, the backedge table automatically contains the shortest occurrence sequences for reconstruction of states. This is not the case, e.g., when using depth-first exploration. When the state space is not explored breadth-first, it is therefore preferable to keep the occurrence sequences in the backedge table short. As an example consider Fig. 4. The occurrence sequences stored in the backedge table for s_4 (state number 5) is $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} s_4$, which is of length 3. A shorter path $s_1 \xrightarrow{b} s_6 \xrightarrow{b} s_4$ has however been found when s_4 was re-discovered from s_6. When re-discovering s_4 from s_6, it is therefore beneficial to replace the backedge $(4, a)$ stored for s_4 to $(3, b)$ such that the shorter occurrence sequence $s_1 \xrightarrow{b} s_6 \xrightarrow{b} s_4$ is stored in the backedge table. It is easy to modify the algorithm to make such simple path optimisations by storing the *depth* of each state in the waiting set along with the full state descriptor and state number. The depth of a state s stored in the waiting set is the length of the occurrence sequence through which s was explored. Whenever a state s is removed from the waiting set in line 7 of Algorithm 2, we obtain the depth d of s. By incrementing d by one, we obtain the depth of each successor state s' of s. If the Reconstruct procedure (see lines 24–30 in Algorithm 2) reconstructs s' based on the backedge table using an occurrence sequences of length greater than $d + 1$, then the backedge stored for s' should be changed to point to the state number of s since going via s results in a shorter occurrence sequence. It is easy to see that the above path optimisation shortens the occurrence sequences stored in the backedge table, but it does not necessary yield the shortest occurrence sequences.

Variant 2: Caching of Full State Descriptors. Another possibility of reducing the time spent on state reconstruction is to maintain a small cache of some full state descriptors for the visited states. As an example, consider Fig. 4 and assume that we have cached state s_3 (with state number 4) during exploration. Then we would not need to do backtracking for state number 4 when we generate state s_5 – we can immediately see that even though states s_3 and s_5 both have the compressed state descriptor h_3, the cached full state descriptor for s_3 is not the same as the full state descriptor for s_5. Caching s_3 also yields an optimisation when we generate state s_4 (with state number 5) when processing s_6. In this case we would not have to backtrack all the way back to the initial state, but as soon as we encounter state number 4 in the backtracking process we can obtain the full state descriptor for s_3 (since it is cached), and it suffices to execute the occurrence sequence $s_3 \xrightarrow{a} s_4$ to reconstruct the full state descriptor for s_4. This shows that caching also optimises state reconstruction for non-cached states. Another way to further optimise backtracking is to re-order the states in the collision lists according to some heuristics that attempt to predict which state is most likely to be re-visited. A simple heuristic is to move a state number to the front of the collision list every time we re-encounter it.

Variant 3: Backwards State Reconstruction. Some modelling languages, including PT-nets and CP-nets, allow transitions to be executed backwards, i.e. we can obtain a function Δ^{-1} such that $\Delta^{-1}(s', t) = s \iff (s, t, s') \in \Delta$. This can be used to execute occurrence sequences from the backedge table backwards, starting from the full state descriptor of a newly generated state s', in order to determine whether s' has already been visited. This has two benefits. Firstly, we do not need to store the occurrence sequence obtained from the backedge table in memory, but can just iteratively look up a backedge in the backedge table and transform the current state using Δ^{-1}. Secondly, the backtracking process may stop early if we encounter an *invalid state*. What qualifies as an invalid state depends on the modelling formalism. A simple implementation for PT-nets and CP-nets is to consider states to be invalid if there is a negative amount of tokens on a place (which may happen when transitions are executed backwards).

Variant 4: Reconstruction of Waiting Set States. In the basic ComBack algorithm we store the full state descriptors for the states in the waiting set. This may take up a considerable amount of memory. It can be observed that we do not actually need to store the full state descriptor for states in the waiting set. It suffices to store the state number as the full state descriptor can be reconstructed from the state number and the backedge table when the state number is selected from the waiting set. This reduces memory usage at the expense of having to make up to $|\mathsf{reach}(s_I)|$ extra reconstructions of states. We can alleviate this, however, if we do depth-first exploration and cache at least the last state that was processed.

Variant 5: Non-deterministic Transitions. For modelling languages with non-deterministic transitions we may have $(s, t, s') \in \Delta \wedge (s, t, s'') \in \Delta$ such that $s' \neq s''$. This means that we may not have a single unique state when executing occurrence sequences obtained from the backedge table, and a state reconstruction procedure is required that operates on sets of states. Consider the reconstruction of a visited state with number n. From the backedge table we obtain (as before) a sequence of backedges $(n_m, t_m) \cdots (n_i, t_i) \cdots (n_2, t_2)(n_1, t_1)$ where $n_1 = 1$ (the initial state). In the i'th step of the reconstruction process when considering the backedge (n_i, t_i), now have a set of states S_1 containing the states that can be reached by executing the transition sequence $t_1 t_2 t_{i-1}$ starting in the initial state. From this set we obtain a new set of states S_2 which is the set of states obtained by executing t_i in those states of S_1 where t_i is enabled. To reduce the size of the set S_2 we observe that S_2 should only contain those states that has the same compressed state descriptor as state number n_{i+1}. The compressed state descriptor for state number n_{i+1} can be obtained from the state table. With a good hash function H, this is expected to keep the size of the sets of states considered during state reconstruction small.

Revised Matches and Reconstruct procedures for Variant 5 are shown in Algorithm 3. The Reconstruct procedure is changed to return a set of

Algorithm 3. MATCHES and RECONSTRUCT procedures for Variant 5

```
1: proc MATCHES($n, s$) is
2:   return $s \in$ RECONSTRUCT($n$)
3:
4: proc RECONSTRUCT($n$) is
5:   if $n = 1$ then
6:     return $\{s_I\}$
7:   else
8:     $(n', t) \leftarrow$ BACKEDGETABLE.LOOKUP($n$)
9:     $S_1 \leftarrow$ RECONSTRUCT($n'$)
10:    $S_2 \leftarrow \{s_2 \in S \mid \exists s_1 \in S_1 : (s_1, t, s_2) \in \Delta\}$
11:    $S_3 \leftarrow \{s_2 \in S_2 \mid n \in$ STATETABLE.LOOKUP($H(s_2)$)$\}$
12:    return $S_3$
```

possible states matching the state number n, so MATCHES is changed to check if s is among those (line 2). The only state corresponding to state number 1 is the initial state (line 6). In line 8 we look up the number of a predecessor state in the backedge table and recursively reconstruct all states that can match that state (line 9). Then we calculate all possible successors of those states (line 10). After that we check that the state number we are looking for, n, is actually in the collision list of the compressed state descriptor of all calculated successors (line 11), and finally return the result. The algorithm will work without the weeding of states in line 11, but at the expense of considering larger state sets.

6 Experimental Results

A prototype of the basic algorithm as described in Sects. 3 and 4 has been implemented in CPN Tools [18] which supports construction and analysis of CPN models [11]. The algorithm is implemented in Standard ML of New Jersey (SML/NJ) [23] version 110.60.

The STATETABLE is implemented as a hash mapping (using lists for handling collisions) and the BACKEDGETABLE is implemented as a dynamic extensible array. This ensures that we can make lookups and insertions in (at least amortized) constant time. The collision list is implemented using SML/NJ's built-in list data type, which is a linked list (rather than an array with a length). A more efficient implementation of the STATETABLE could be obtained using very tight hashing [5]. This would allow us to remove some redundant bits from the compressed state descriptor. We have implemented both depth-first exploration (DFS) and breadth-first exploration (BFS).

The compressed state descriptors calculated by the hash function as well as the state numbers are 31-bit unsigned integers as SML/NJ uses the 32nd bit for garbage collection. The hash function used is defined inductively on the state of the CPN model. In CP-nets, a state of the system is a *marking* of a set of *places*. Each marking is a *multi-set* over a given *type*. We use a standard hash function for each type. We extend this hash function to multi-sets by using a *combinator* *function*, which takes two hash values and returns a new hash value. We extend the hash functions on markings of places to a hash function of the entire model by using the combinator function on the place hash functions.

We also implemented caching of full state descriptors as explained in Sect. 5. The caching strategy used is simple: we use a hash mapping from state numbers to full state descriptors, which does not account for collisions of hash values. That way, if we allocate a hash mapping of, say, size 1000, we can store at most 1000 full state descriptors in the cache. We have not implemented re-ordering of states in the collision lists, as the collision lists have length at most 2 (with two exceptions) for all our examples.

We use a test-suite consisting of three kinds of models: small examples, medium-sized examples and real-life applications. In the first category, we have three models: a model of the dining philosophers system (DP), a model of replicating database managers (DB), and a model of a stop-and-wait network protocol (SW). In the second category, we have a model of a telephone system (TS). In the last category, we have a model of a protocol (ERDP) for distributing network prefixes to gateways in a network consisting of standard wired networks and wireless mobile ad-hoc networks [14]. All of the models are parametrised: DP by the number of philosophers, DB by the number of database managers, SW by the number of packets transmitted and the capacity of the network, TS by the number of telephones, and ERDP by the number of available prefixes and the capacity of the network. We will denote each model by its name and its parameter(s), e.g. DP22 denotes DP with 22 philosophers and ERDP6,2 denotes the ERDP protocol with six prefixes and a network capacity of two.

We have evaluated the performance of the ComBack method without cache, denoted by ComBack, and with cache of size n, denoted ComBack n. We have compared the ComBack method with implementations of basic hash compaction [27], bit-state hashing [8] by means of double hashing [4] which uses a linear combination of two hash functions to compute, in this case, 15 compressed state descriptors. Instead of storing the compressed state descriptors, like hash compaction, bit-state hashing uses the values to set bits in a bit-array. Finally, we compare the ComBack method to standard state space exploration of the full state space using a hash table for storing the full state descriptors. For each model, we have measured how much memory and how much CPU time was used to conduct the state space exploration. Memory is measured by performing a full garbage collection and measuring the size of the heap. This is done every 0.5 second or 40 states, whichever comes last. As garbage collection takes time, the CPU time used is measured independently. We have measured the time three times and used the average as the result.

Table 1 shows the results of the experiments. For each model (column 1) and each exploration method (column 2), we show the number of nodes (states) and arcs explored (columns 3 and 4). We also show the CPU time spent (in seconds) and the amount of space (memory) used (in mega-bytes) for a depth-first traversal (DFS) and a breadth-first traversal (BFS) of the state space (columns 5, 7, 10, and 12). In addition we show how much time and memory is used relative to traversal using a standard exploration using DFS (columns 6, 8, 11, and 13) and how much memory (in bytes) is used per state (columns 9 and 14).

We note that for each model, independent of the reduction technique, either DFS performs better memory-wise than BFS or vice versa. For the more realistic examples, TS and ERDP, DFS is slower than BFS. This is due to the fact that the models resemble real systems and have more complex behaviour, which leads to very long occurrence sequences in the backedge table, and thus impacts the performance of the ComBack method. If we instrument the ComBack method with even a small cache when using DFS, or if we use BFS, processing is much faster for realistic examples. We see that the ComBack method uses quite a bit more memory than the 5 machine words predicted in the previous section. One cause for this is that the calculation in Sect. 4 did not take the WaitingSet into account and only considered the *elements* of the state table and the backedge table, not the tables themselves. Furthermore, SML is not very memory-efficient, doubling usage. We also note that the standard exploration as well as the ComBack method using BFS were not able to complete due to lack

Table 1. Experimental results

model	method	nodes	arcs	DFS					BFS				
				time		space			time		space		
				sec	%	Mb	%	/state	sec	%	Mb	%	/state
DP22	ComBack	39604	481625	2791	10337	23.0	97	608	59	219	9.8	42	260
	ComBack 100	39604	481625	800	2963	23.0	98	610	56	207	9.9	42	261
	ComBack 1000	39604	481625	98	363	23.6	100	625	57	211	10.6	45	281
	Hash compaction	39603	481609	25	93	20.8	88	550	26	96	8.4	35	222
	Bit-state	39604	481609	28	104	32.0	135	846	29	107	20.0	85	531
	Standard	39604	481625	27	100	23.6	100	625	27	100	14.3	61	380
DB9	ComBack	59051	314947	60	214	4.5	10	80	63	225	11.9	28	212
	ComBack 100	59051	314947	50	178	4.7	11	83	51	182	12.1	28	214
	ComBack 1000	59051	314947	48	171	5.5	13	98	44	157	12.9	30	229
	Hash compaction	59049	314937	25	89	1.4	3	25	27	96	10.1	23	179
	Bit-state	59051	314947	29	104	12.3	28	218	33	118	21.3	49	379
	Standard	59051	314947	28	100	43.3	100	769	28	100	43.4	100	770
DB10	ComBack	196832	1181001	286	44	15.4	9	82	307	48	43.1	25	230
	ComBack 100	196832	1181001	247	38	15.6	9	83	264	41	43.3	25	231
	ComBack 1000	196832	1181001	240	37	16.6	10	89	250	39	44.4	26	236
	Hash compaction	196798	1180790	118	18	4.9	3	26	133	21	36.8	21	196
	Bit-state	196832	1181001	138	21	12.3	7	66	152	24	46.3	27	247
	Standard	196832	1181001	643	100	174.0	100	927	693	106	174.0	100	927
SW7,4	ComBack	215196	1242386	115	319	17.5	41	85	115	319	20.1	47	98
	ComBack 100	215196	1242386	68	189	17.6	41	86	100	278	20.2	47	98
	ComBack 1000	215196	1242386	64	178	17.9	42	87	93	258	20.6	48	100
	Hash compaction	214569	1238803	33	92	5.2	12	25	37	103	9.8	23	48
	Bit-state	215196	1242386	41	114	12.3	28	60	46	128	18.3	43	89
	Standard	215196	1242386	36	100	43.0	100	210	40	100	43.1	100	210
TS5	ComBack	107648	1017490	3302	6115	51.4	84	500	103	191	17.6	29	172
	ComBack 100	107648	1017490	933	1728	51.4	83	501	102	189	17.7	29	172
	ComBack 1000	107648	1017490	207	383	51.9	85	506	107	198	18.4	30	180
	Hash compaction	107647	1017474	50	93	45.7	75	445	52	96	14.7	24	143
	Bit-state	107648	1017490	58	107	55.4	90	540	62	115	25.8	42	251
	Standard	107648	1017490	54	100	61.2	100	596	57	106	45.0	73	438
ERDP6,2	ComBack	207003	1199703	986	865	29.1	33	147	867	761	35.7	41	181
	ComBack 100	207003	1199703	259	227	29.0	33	147	481	422	35.8	41	181
	ComBack 1000	207003	1199703	205	180	29.6	34	150	402	353	36.4	42	184
	Hash compaction	206921	1199200	106	93	5.1	6	26	114	100	18.6	21	94
	Bit-state	207003	1199703	123	108	12.3	14	62	135	118	27.3	31	138
	Standard	207003	1199703	114	100	87.4	100	443	131	115	88.5	101	449
ERDP6,3	ComBack	4277126	31021101	42711	-	572.3	-	140	65354	-	708.1	-	174
	ComBack 100	4277126	31021101	18043	-	571.2	-	140	-	-	-	-	-
	ComBack 1000	4277126	31021101	23084	-	571.7	-	140	-	-	-	-	-
	Hash compaction	4270926	30975030	3341	-	113.5	-	28	20512	-	403.6	-	99
	Bit-state	4277125	31021091	3732	-	12.1	-	3	17481	-	347.9	-	85
	Standard	-	-	-	-	-	-	-	-	-	-	-	-

of memory for the ERDP6,3 model. The hash compaction bit-state hashing were also not able to explore all states for this example (as can be seen in the nodes column of Table 1). This means we are comparing methods guaranteeing full coverage with methods that do not, so while the hash compaction and bit-state hashing methods seem to perform well, they do so at a cost.

Figure 5 shows charts depicting memory and time usage relative to standard DFS exploration (i.e. one chart for columns 5 and 7 and another chart for columns 10 and 12). These charts allow us to better understand how the different exploration methods perform compared to each other, independent of the example. We see that the values fall into 7 rectangles corresponding to 6 different exploration methods and an abnormal experiment. Rectangle 1: standard exploration; all results are near 100% on both axes, showing that when we store full state descriptors in a hash table, it does not matter whether we use DFS or BFS. Rectangle 2: hash compaction; all are near 100% on the time axis and between 2% and 100% (DFS) or 20% and 40% (BFS) on the memory axis, showing that hash compaction uses as much time as storing the full state descriptors, but significantly less space. Rectangle 3: bit-state hashing; all are near 100% time-wise, but slightly higher than 1 and 2 (this is probably because we have to calculate two hash values instead of just one). All range between 15% and 150% memory-wise. The bit-state hashing method consistently uses 12.5 mega-bytes plus the size of the waiting set, so it performs well memory-wise on models with large state spaces, but performs poorly on models with small state spaces. This means that memory optimisations are possible, but customisation is required by the user. All the show models are reasonable large, leading to reasonable performance of bit-state hashing. Rectangle 4: ComBack without cache; all are above 150% time-wise and between 10% and 100% (DFS) or 25% and 40% (BFS) memory-wise. Time is also better bounded in the BFS results. This indicates that ComBack without cache yields a reduction (it is never above 100%), and when using BFS we have better control of the time and memory used. DFS makes it possible to save more memory, but can be very costly time-wise, and sometimes we do not save any memory at all (e.g. in the Dining philosophers example, where we can end up with most of the state space in the waiting set). Rectangles 5+6: ComBack with cache; these use slightly more memory but less time than 4, in particular in the DFS case. More cache yields more memory and less time used, but the differences are not that large, and even a small cache yields great optimizations in time compared to the ComBack method with no cache at all. Rectangle 7: DB10; these points fall outside of all the other boxes. Inspection of the data in Table 1 shows that the DB10 example has irregular behaviour, as exploration using the standard exploration is slow as a full state descriptor for this model is large, and thus the SML/NJ garbage collector is invoked often. This yields a performance penalty and causes all other experiments, as they are relative to the standard exploration, to fall outside the other boxes. ERDP6,3 is not shown as the standard exploration was unable to terminate.

All of the shown experiments have been performed using a hash-function generating 31-bit compressed state descriptors. We have also tested the method

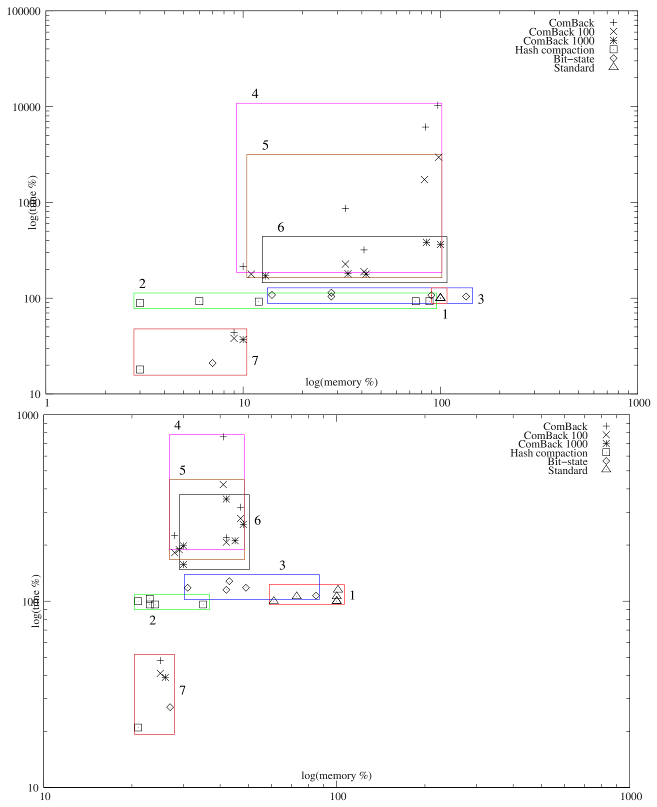


Fig. 5. Time and memory usage for the various reduction techniques using DFS exploration (top) and BFS exploration (bottom). Values are relative to corresponding values for standard depth-first exploration.

using a hash function generating 62-bit compressed state descriptors, but have not shown those results, as the time usage is the same but more memory is consumed, as the 31-bit hash function causes few collisions. We have verified the quality of the hash-function by calculating the lengths of the collision lists for all examples. The worst case is example SW7,4, where there are 214009 collision lists of length 1, 592 lists of length 2 and 1 list of length 3, so 99.7 % of the collision lists have the minimum length. It also means that hash compaction misses at least $1 \cdot 592 + 2 \cdot 1 = 594$ states due to hash collisions.

7 Conclusions and Future Work

In this paper we have presented the ComBack method for alleviating the state explosion problem. The basic idea of the method is to augment the hash compaction method with a backedge table that makes it possible to reconstruct full state descriptors and ensure full coverage of the state space. We have made a prototype implementation of the method in CPN Tools and our experimental results demonstrate that the method (as expected) uses more time and memory than hash compaction, but less memory than ordinary state space exploration.

The advantage of the ComBack method is that it guarantees full coverage of the state space, unlike related methods such as hash compaction and bit-state hashing. From a practical viewpoint one could therefore use methods such as hash compaction in early phases of a verification process to discover errors, and when no further errors can be detected, the ComBack method could be used for formal verification of properties.

In this paper we have not discussed verification of properties using the ComBack method. It can be observed that the method explores the full state space without mandating a particular exploration order. Furthermore, the state reconstruction that occurs when checking whether a state has already been visited can be made fully transparent to the verification algorithm being applied in conjunction with the state space exploration. This makes the method compatible with most on-the-fly verification algorithms (e.g., verification of safety properties and on-the-fly LTL model checking [26]). The ComBack method is also compatible with off-line verification algorithms such as CTL model checking [15] since the backedge table allows the reconstruction of any of the full state descriptors which in turn allows the forward edges between states to be reconstructed. Alternatively, we can simple store the forward edges in an additional table during state space exploration.

The ComBack method opens up several areas for future work. One topic is the integration of verification algorithms as sketched in the previous paragraph. Future work also includes implementation of the additional variants presented in Sect. 5, and the development and evaluation of caching strategies and organisation of collision lists to reduce the time spent on state reconstruction. It would also be interesting to compare the ComBack method to other complete techniques such as state caching [6]. Another important topic is to explore the combination of the ComBack method with other reduction methods. For this purpose, partial-order methods [17, 24] appear particularly promising as they reduce the in-degree of states which in turn will lead to a reduction in the number of state reconstructions.

References

1. Behrmann, G., Larsen, K.G., Pelánek, R.: To Store or Not to Store. In: Proc. of CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
2. Bryant, R.E.: Graph Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
3. Christensen, S., Kristensen, L.M., Mailund, T.: A Sweep-Line Method for State Space Exploration. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
4. Dillinger, P.C., Manolios, P.: Fast and accurate Bitstate Verification for SPIN. In: Graf, S., Mounier, L. (eds.) Proc. of SPIN 2004. LNCS, vol. 2989, Springer, Heidelberg (2004)
5. Geldenhuys, J., Valmari, A.: A Nearly Memory-Optimal Data Structure for Sets and Mappings. In: Ball, T., Rajamani, S.K. (eds.) Proc. of SPIN 2003. LNCS, vol. 2648, pp. 136–150. Springer, Heidelberg (2003)

6. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-Space Caching Revisited. Formal Methods in System Design 7(3), 227–241 (1995)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
8. Holzmann, G.J.: An Analysis of Bitstate Hashing. Formal Methods in System Design 13, 289–307 (1998)
9. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, London (2004)
10. Ip, C.N., Dill, D.L.: Better Verification Through Symmetry. Formal Methods in System Design, vol. 9 (1996)
11. Jensen, K.: Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. In: Basic Concepts, vol. 1, Springer, Heidelberg (1992)
12. Jensen, K.: Condensed State Spaces for Symmetrical Coloured Petri Nets. Formal Methods in System Design, vol. 9 (1996)
13. Kam, T.: State Minimization of Finite State Machines using Implicit Techniques. PhD thesis, University of California at Berkeley (1995)
14. Kristensen, L.M., Jensen, K.: Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) Proc. of INT'04. LNCS, vol. 3147, pp. 248–269. Springer, Heidelberg (2004)
15. Kupferman, O., Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Branching-Time Model Checking. Journal of the ACM 47(2), 312–360 (2000)
16. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
17. Peled, D.: All for One, One for All: On Model Checking Using Representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
18. 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)
19. Reisig, W.: Petri Nets. In: EATCS Monographs on Theoretical Computer Science, vol. 4, Springer, Heidelberg (1985)
20. Schmidt, K.: LoLA - A Low Level Analyser . In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000)
21. Stern, U., Dill, D.L.: Improved Probabilistic Verification by Hash Compaction. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995)
22. Stern, U., Dill, D.L.: Using Magnetic Disk instead of Main Memory in the Murphi Verifier. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
23. Ullman, J.D.: Elements of ML Programming. Prentice-Hall, Englewood Cliffs (1998)
24. Valmari, A.: Stubborn Sets for Reduced State Space Generation. In: Advances in Petri Nets '90. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1990)
25. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
26. Vardi, M., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Proc. of IEEE Symposium on Logic in Computer Science, pp. 322–331 (1986)
27. Wolper, P., Leroy, D.: Reliable Hashing without Collision Detection. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)

Computing Minimal Elements of Upward-Closed Sets for Petri Nets

Hsu-Chun Yen[⋆] and Chien-Liang Chen

Dept. of Electrical Engineering, National Taiwan University
Taipei, Taiwan 106, Republic of China
yen@cc.ee.ntu.edu.tw

Abstract. *Upward-closed sets* of integer vectors enjoy the merit of having a finite number of *minimal elements*, which is behind the decidability of a number of Petri net related problems. In general, however, such a finite set of minimal elements may not be effectively computable. In this paper, we develop a unified strategy for computing the sizes of the minimal elements of certain upward-closed sets associated with Petri nets. Our approach can be regarded as a refinement of a previous work by Valk and Jantzen (in which a necessary and sufficient condition for effective computability of the set was given), in the sense that complexity bounds now become available provided that a bound can be placed on the size of a witness for a key query. The sizes of several upward-closed sets that arise in the theory of Petri nets as well as in backward-reachability analysis in automated verification are derived in this paper, improving upon previous decidability results shown in the literature.

1 Introduction

A set U over k-dimensional vectors of natural numbers is called *upward-closed* (or *right-closed*) if $\forall x \in U, y \geq x \Longrightarrow y \in U$. It is well known that an upward-closed set is completely characterized by its *minimal* elements, which always form a finite set. Aside from being of interest mathematically, evidence has suggested that upward-closed sets play a key role in a number of decidability results in automated verification of *infinite state systems* [1,4,6,13]. In the analysis of Petri nets, the notion of upward-closed sets is closely related to the so-called *property of monotonicity* which serves as the foundation for many decision procedures for Petri net problems. What the monotonicity property says is that if a sequence σ of transitions of a Petri net is executable from a marking (i.e., configuration) $\mu \in \mathbb{N}^k$, then the same sequence is legitimate at any marking greater than or equal to μ. That is, all the markings enabling σ form an upward-closed set.

In spite of the fact that the set of all the minimal elements of an upward-closed set is always finite, such a set may not be effectively computable in general. There are, however, certain interesting upward-closed sets for which their

[⋆] Corresponding author. Research supported in part by '*Excellent Research Projects of National Taiwan University*,' 95R0062-AE00-05.

minimal elements are effectively computable. A notable example is the set of initial markings of a Petri net from which a designated final marking is *coverable*. More recent work of [1] demonstrated decidability to compute, from a given upward-closed set of final states, the set of states that are *backward reachable* from the final states. In a practical aspect of research in upward-closed sets, *BDD*-like data structures called *covering sharing trees* [6] have been developed to represent in a compact way collections of upward-closed sets over numerical domains. Subsequent experimental results [7] have further demonstrated such symbolic representations and manipulations of upward-closed sets to be promising in performing backward reachability analysis of problems for infinite state systems such as Petri nets.

Given the importance of upward-closed sets, it is of interest theoretically and practically to be able to characterize the class of upward-closed sets for which their minimal elements are computable. Along this line of research, Valk and Jantzen ([13]) presented a sufficient and necessary condition under which the set of minimal elements of an upward-closed set is guaranteed to be effectively computable. Supposed U is an upward-closed set over $\mathbb{N}^k$ and ω is a symbol representing something being arbitrarily large. In [13], it was shown that the set of minimal elements of U is effectively computable iff the question '$reg(v) \cap U \neq \emptyset$?' is decidable for every $v \in (\mathbb{N} \cup \{\omega\})^k$, where $reg(v) = \{x \mid x \in \mathbb{N}^k, x \leq v\}$. (More will be Such a strategy has been successfully applied to showing computability of a number of upward-closed sets associated with Petri nets ([13]). Note, however, that [13] reveals no complexity bounds for the sizes of the minimal elements. As knowing the size of minimal elements might turn out to be handy in many cases, the following question arises naturally. If more is known about the query '$reg(v) \cap U \neq \emptyset$?' (other than just being decidable), could the size of the minimal elements be measured? In fact, answering the question in the affirmative is the main contribution of this work.

Given a vector $v \in (\mathbb{N} \cup \omega)^k$, suppose $||v||$ is defined to be the maximum component (excluding ω) in v. We demonstrate that for every v, if a bound on the size of a witness for $reg(v) \cap U \neq \emptyset$? (if one exists) is available, then such a bound can be applied inductively to obtain a bound for *all* the minimal elements of U. In a recent article [14], such a strategy was first used for characterizing the solution space of a restricted class of *parametric timed automata*. In this paper, we move a step further by formulating a general strategy as well as applying our unified framework to a wide variety of Petri net problems with upward-closed solution sets. In addition to those upward-closed sets investigated in [13] for general Petri nets, we illustrate the usefulness of our approach in performing *backward-reachability* analysis, which is a useful technique in automated verification.

Given a system S with initial state q, and a designated set of states Q, *backward-reachability analysis* involves computing the set $pre^*(S, Q)$ which consists of all the states from which some state in Q is reachable, and then deciding whether $q \in pre^*(S, Q)$. *Forward-reachability analysis*, on the other hand, computes all the states that can be reached from q to see whether the intersection with Q is non-empty or not. In general, $pre^*(S, Q)$ may not be computable for infinite state systems. We show that for certain classes of Petri nets and state set Q, $pre^*(S,Q)$ is not only upward-closed but falls into the category to which our unified approach can be applied. Such Petri nets include several well known subclasses for which reachability is characterizable by *integer linear programming.* Our analysis can also be applied to the model of *lossy* vector addition systems with states (VASSs) [4] to derive bounds for the backward-reachability sets. For (conventional or lossy) VASSs, we further enhance the work of [4] by providing complexity bounds for the so-called *global model checking problem* with respect to a certain class of formulas. (In [4], such a problem was only shown to be decidable, yet no complexity was available there.) Upward-closed sets associated with a kind of *parametric clocked Petri nets* are also investigated in this paper, serving as yet another application of our unified approach.

The remainder of this paper is organized as follows. Section 2 gives basic definitions and notations of upward-closed sets, as well as the models of Petri nets and vector addition systems with states. In Section 3, our unified strategy for reasoning about the sizes of upward-closed sets is investigated. Section 4 is devoted to applications of our strategy to various problems in Petri nets and related models. A conclusion is given in Section 5.

2 Preliminaries

Let $\mathbb{Z}$ ($\mathbb{N}$, resp.) be the set of all integers (nonnegative integers, resp.), and $\mathbb{Z}^k$ ($\mathbb{N}^k$, resp.) be the set of k-dimensional vectors of integers (nonnegative integers, resp.). Let $\mathbf{0}$ be the *zero vector*. Let $v(i), 1 \le i \le k$, denote the i-th component of a k-dimensional vector v. Given two vectors u and $v(\in \mathbb{N}^k)$, $u \le v$ if $\forall 1 \le i \le k, u(i) \le v(i)$, and $u < v$ if $u \le v$ and $u \ne v$. We define the *max-value* of v, denoted by $||v||$, to be $max\{|v(i)| \mid 1 \le i \le k\}$, i.e., the absolute value of the largest component in v. For a set of vectors $V = \{v_1, ..., v_m\}$, the *max-value* of V (also written as $||V||$) is defined to be $max\{||v_i|| \mid 1 \le i \le m\}$. In our subsequent discussion, we let $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$ (ω is a new element capturing the notion of something being 'arbitrarily large'). We assume the following arithmetics for ω: (1) $\forall n \in \mathbb{N}, n < \omega$, (2) $\forall n \in \mathbb{N}_\omega, n + \omega = \omega - n = \omega$, $(n+1) \times \omega = \omega, 0 \times \omega = \omega \times 0 = 0$. We also let $\mathbb{N}^k_\omega = (\mathbb{N} \cup \{\omega\})^k = \{(v_1, ..., v_k) \mid v_i \in (\mathbb{N} \cup \{\omega\}), 1 \le i \le k\}$. For a $v \in \mathbb{N}^k_\omega$, we also write $||v||$ to denote $max\{v(i) \mid v(i) \ne \omega\}$ (i.e., the largest component in v excluding ω) if $v \ne (\omega, ..., \omega)$; $||(\omega, ..., \omega)|| = 1$. For an element $v \in \mathbb{N}^k_\omega$, let $reg(v) = \{w \in \mathbb{N}^k \mid w \le v\}$.

A set $U(\subseteq \mathbb{N}^k)$ is called *upward-closed* (or *right-closed* in some literature) if $\forall x \in U$, $\forall y, y \ge x \implies y \in U$. An element x $(\in U)$ is said to be *minimal* if there is no y $(\ne x)$ $\in U$ such that $y < x$. We write $min(U)$ to denote the set of minimal elements of U. From Dicksons lemma, it is well-known that for each upward-closed set $U(\subseteq \mathbb{N}^k)$, $min(U)$ is finite. Even so, $min(U)$ might not be effectively computable in general.

Given a function f, we write the k-fold composition of f as $f^{(k)}$ (i.e., $f^{(k)}(x) = \overbrace{f \circ \cdots \circ f}^{k}(x)$). Given a set T, we denote by T^* (resp., T^ω) the set of all finite (resp., infinite) strings of symbols from T, and $T^+ = T^* - \{\lambda\}$, where λ is the empty string.

A *Petri net* (*PN*, for short) is a 3-tuple $\mathcal{P} = (P, T, \varphi)$, where P is a finite set of *places*, T is a finite set of *transitions*, and φ is a *flow function* $\varphi : (P \times T) \cup (T \times P) \rightarrow N$. Let k and m denote $|P|$ (the number of places) and $|T|$ (the number of transitions), respectively. The k is also called the *dimension* of the PN. A *marking* is a mapping $\mu : P \rightarrow N$. The *transition vector* of a transition t, denoted by $\bar{t}$, is a k-dimensional vector in $\mathbb{Z}^k$, such that $\bar{t}(i) = \varphi(t, p_i) - \varphi(p_i, t)$, and the set of transition vectors, denoted by $\bar{T}$, to be $\{\bar{t} \mid t \in T\}$.

A transition $t \in T$ is *enabled* at a marking μ iff $\forall p \in P$, $\varphi(p, t) \leq \mu(p)$. If a transition t is enabled, it may *fire* and yields marking μ' (written as $\mu \xrightarrow{t} \mu'$) with $\mu'(p) = \mu(p) - \varphi(p, t) + \varphi(t, p)$, $\forall p \in P$. A sequence of transitions $\sigma = t_1...t_n$ is a *firing sequence* from μ_0 iff $\mu_0 \xrightarrow{t_1} \mu_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \mu_n$ for some markings μ_1,...,μ_n. (We also write '$\mu_0 \xrightarrow{\sigma} \mu_n$'.) We write '$\mu_0 \xrightarrow{\sigma}$' to denote that σ is enabled and can be fired from μ_0, i.e., $\mu_0 \xrightarrow{\sigma}$ iff there exists a marking μ such that $\mu_0 \xrightarrow{\sigma} \mu$. The notation $\mu_0 \xrightarrow{*} \mu$ is used to denote the existence of a $\sigma \in T^*$ such that $\mu_0 \xrightarrow{\sigma} \mu$. A *marked* PN is a pair $((P, T, \varphi), \mu_0)$, where (P, T, φ) is a PN, and μ_0 is called the *initial marking*. Throughout the rest of this paper, the word 'marked' will be omitted if it is clear from the context. By establishing an ordering on the elements of P and T (i.e., $P = \{p_1, ..., p_k\}$ and $T = \{r_1, ..., r_m\}$), we can view a marking μ as a k-dimensional vector with its *i-th* component being $\mu(p_i)$, and $\#_\sigma$ as an m-dimensional vector with its jth entry denoting the number of occurrences of transition r_j in σ. The *reachability set* of $\mathcal{P}$ with respect to μ_0 is the set $R(\mathcal{P}, \mu_0) = \{\mu \mid \exists \sigma \in T^*, \mu_0 \xrightarrow{\sigma} \mu\}$. The *reachability problem* is that of, given a marked PN ($\mathcal{P}$, μ_0) and a marking μ, deciding whether $\mu \in R(\mathcal{P}, \mu_0)$. $F(\mathcal{P}, \mu_0)$ $(= \{\sigma \in T^* \mid \mu_0 \xrightarrow{\sigma}\})$ denotes the set of all fireable sequences of transitions in PN $(\mathcal{P}, \mu_0)$. Given a $\sigma \in T^\omega$, $In(\sigma)$ denotes the set of all elements in T that occur infinitely many times in σ.

A *k-dimensional vector addition system with states* (VASSs) is a 5-tuple (v_0, V, s_1, S, δ), where $v_0 \in \mathbb{N}^k$ is called the *start vector*, V $(\subseteq \mathbb{Z}^k)$ is called the set of *addition rules*, S is a finite set of *states*, $\delta(\subseteq S \times S \times V)$ is the transition relation, and s_1 $(\in S)$ is the *initial state*. Elements (p, q, v) of δ are called *transitions* and are usually written as $p \rightarrow (q, v)$. A configuration of a VASS is a pair (p, x) where $p \in S$ and $x \in \mathbb{N}^k$. For convenience, we write $state((p, x)) = p$ and $val((p, x)) = x$. (s_1, v_0) is the *initial configuration*. The transition $p \rightarrow (q, v)$ can be applied to the configuration (p, x) and yields the configuration $(q, x + v)$, provided that $x + v \geq \mathbf{0}$. In this case, $(q, x + v)$ is said to *follow* (p, x). Let σ_0 and σ_t be two configurations. Then σ_t is said to be *reachable* from σ_0 iff $\sigma_t = \sigma_0$ or configurations $\sigma_1, ..., \sigma_{t-1}$ such that σ_{r+1} follows σ_r for $r = 0, ..., t - 1$. A k-dimensional VASS is basically a k-counter machine without zero-test capabilities (i.e., the counter machine cannot test individual counters for empty and then acts accordingly). It is well known that PNs and VASSs are computationally equivalent.

3 A Strategy for Computing the Sizes of Minimal Elements

It is well known that every upward-closed set over $\mathbb{N}^k$ has a finite number of minimal elements. However, such a finite set may not be effectively computable in general. In an article [13] by Valk and Jantzen, the following result was proven which suggests a sufficient and necessary condition under which the set of minimal elements of an upward-closed set is effectively computable:

Theorem 1. *([13]) For each upward-closed set $K (\subseteq \mathbb{N}^k)$, $min(K)$ is effectively computable iff for every $v \in \mathbb{N}^k_\omega$, the problem '$reg(v) \cap K \neq \emptyset$?' is decidable. (Recall that $reg(v) = \{w \in \mathbb{N}^k \mid w \leq v\}$.)*

In what follows, we show that for every $v \in \mathbb{N}^k_\omega$, should we be able to compute the size of a witness for $reg(v) \cap U \neq \emptyset$ (if one exists), then an upper bound can be placed on the sizes of all the minimal elements.

Theorem 2. *Given an upward-closed set $U (\subseteq \mathbb{N}^k)$, if for every $v \in \mathbb{N}^k_\omega$, a witness $\hat{w} \in \mathbb{N}^k$ for '$reg(v) \cap U \neq \emptyset$' (if one exists) can be computed with*

(i) $||\hat{w}|| \leq b$, for some $b \in \mathbb{N}$ when $v = (\omega, ..., \omega)$,
(ii) $||\hat{w}|| \leq f(||v||)$ when $v \neq (\omega, ..., \omega)$, for some monotone function f,

then $||min(U)|| \leq f^{(k-1)}(b)$.

Proof. Given an arbitrary $h, 1 \leq h \leq k$, we show inductively that for each $m \in min(U)$, there exist h indices $i_{m_1}, ..., i_{m_h}$ such that $\forall l, 1 \leq l \leq h$, $m(i_{m_l}) \leq f^{(h-1)}(b)$.

(*Induction Basis*). Consider the case when $h = 1$. We begin with $v_0 = (\omega, ..., \omega)$. Assume that w_0 is a witness for $reg(v_0) \cap U \neq \emptyset$, which, according to the assumption of the theorem, satisfies $||w_0|| \leq b = f^{(0)}(b)$. Let $min_1(U) = min(U) \backslash reg((b, ..., b))$, i.e., those in $min(U)$ that have at least one component larger than b. If $min_1(U) = \emptyset$, then the theorem follows since $f^{(0)}(b)$ becomes a bound for $||min(U)||$. Otherwise, $\forall m \in min_1(U)$, it must be the case that $\exists i,\ 1 \leq i \leq k, m(i) < b$; otherwise, m would not have been minimal since $w_0 \leq (b, ..., b)$. Hence, the assertion holds for $h = 1$, i.e., $f^{(0)}(b)$ is a bound for at least one component for all the elements in $min(U)$. See Figure 1.

(*Induction Step*). Assume that the assertion holds for h ($< k$), we now show the case for $h + 1$. Consider $min_h(U) = min(U) \backslash \bigcup_{v \in N^k, ||v|| \leq f^{(h-1)}(b)} \{reg(v)\}$, i.e., the set of minimal elements that have at least one coordinate exceeding $f^{(h-1)}(b)$. If $min_h(U) = \emptyset$, the assertion holds; otherwise, take an arbitrary $m \in min_h(U)$, and let $i_{m_1}, ..., i_{m_h}$ be the indices of those components satisfying the assertion, i.e., $\forall 1 \leq j \leq h, m(i_{m_j}) \leq f^{(h-1)}(b)$. Let v^m_h be such that $v^m_h(l) = m(i_{m_j})$, if $l = i_{m_j}$; $=\omega$ otherwise. That is, v^m_h agrees with m on coordinates $i_{m_1}, ..., i_{m_h}$, and carries the value ω for the remaining coordinates. Notice that $||v^m_h|| \leq f^{(h-1)}(b)$. According to the assumption of the theorem, a witness w^m_h for $reg(v^m_h) \cap U \neq \emptyset$ with $||w^m_h||$ bounded by $f(||v^m_h||)$ ($\leq f(f^{(h-1)}(b))$) can be obtained. Furthermore, $w^m_h(i_{m_j}) \leq m(i_{m_j})$, $1 \leq j \leq h$. (Notice that $m \in$

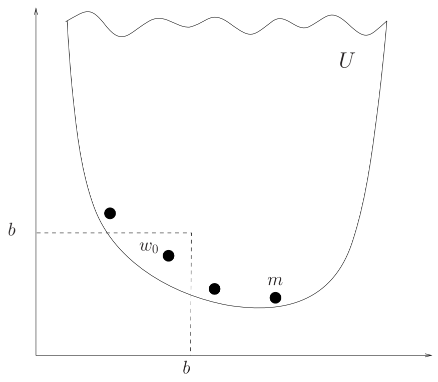


Fig. 1. Induction Basis

$min_h(U)$ implies the existence of such a witness.) It must be the case that there exists an index $i_{m_{h+1}} (\notin \{i_{m_1}, ..., i_{m_h}\})$ such that $m(i_{m_{h+1}}) \leq w_h^m(i_{m_{h+1}})$ ($\leq f(f^{(h-1)}(b))) = f^{(h)}(b)$), since otherwise, $w_h^m < m$ – contradicting that m being minimal. The induction step is therefore proven. ▮

4 Some Applications

In this section, Theorem 2 is used to derive upper bounds for the minimal elements of various upward-closed sets associated with PNs and VASSs.

4.1 Petri Nets

We examine some upward-closed sets defined and discussed in [13]. Some definitions from [13] are recalled first. Given a PN (P, T, φ), a vector $\mu \in \mathbb{N}^k$ is said to be

- $\hat{T}$*-blocked*, for $\hat{T} \subseteq T$, if $\forall \mu' \in R(\mathcal{P}, \mu)$, $\neg(\exists t \in \hat{T}, \mu' \xrightarrow{t})$. For the case when $\hat{T} = T$, μ is said to be a *total deadlock*.
- *dead* if $F(\mathcal{P}, \mu)$ is finite.
- *bounded* if $R(\mathcal{P}, \mu)$ is finite; otherwise, it is called *unbounded*.
- $\hat{T}$*-continual*, for $\hat{T} \subseteq T$, if there exists a $\sigma \in T^\omega$, $\mu \xrightarrow{\sigma}$ and $\hat{T} \subseteq In(\sigma)$.

For a PN $(\mathcal{P}, \mu_0)$, consider the following four sets defined in [13]:

- *NOTBLOCKED*($\hat{T}$)=$\{\mu \in \mathbb{N}^k \mid \mu$ is not $\hat{T}$-*blocked*$\}$.
- *NOTDEAD*=$\{\mu \in \mathbb{N}^k \mid \mu$ is not *dead*$\}$.
- *UNBOUNDED*=$\{\mu \in \mathbb{N}^k \mid \mu$ is *unbounded*$\}$.
- *CONTINUAL*($\hat{T}$)=$\{\mu \in \mathbb{N}^k \mid \mu$ is $\hat{T}$-*continual*$\}$.

It has been shown in [13] that for each of the above four upward-closed sets, the '$reg(v) \cap K \neq \emptyset$?' query of Theorem 1 is decidable; as a consequence, the set of minimal elements is effectively computable. We now show how to use Theorem 2 to estimate the bound of the minimal elements for each of the four sets. To this end, we show that if '$reg(v) \cap K \neq \emptyset$', where K is any of the above four upward-closed sets, then there is a witness whose max-value is bounded by $n^{2^{d \times k \times logk}}$, where d is a constant, n is the maximum number of tokens that can be added to or subtracted from a place in the k-dimensional PN and n is independent of v.

Our analysis makes use of a *multi-parameter* version [12] of the inductive strategy developed by Rackoff in [11] in which the complexities of the boundedness and covering problems for vector addition systems (equivalently, PNs) were developed. By a multi-parameter analysis of PNs we mean treating the *dimension* and the *max-value* of PN transition vectors as two separate parameters, as opposed to regarding the *size* of the PN as the sole parameter as was the case in [11]. (The interested reader is referred to [12] for more about the multi-parameter approach for analyzing PNs.) By doing so, the key function f in Theorem 2 is better estimated. Before going into details, we require the following definitions, most of which can be found in [11]. Intuitively speaking, Rackoff's strategy relies on showing that if a path exhibiting unboundedness or coverability exists, then there is a 'short' witness.

A *generalized marking* is a mapping $\mu : P \rightarrow \mathbb{Z}$ (i.e., negative coordinates are allowed). A $w \in \mathbb{Z}^k$ is called *i-bounded* (resp., *i-r bounded*) if $0 \leq w(j)$, $\forall 1 \leq j \leq i$ (resp. $0 \leq w(j) \leq r$, $\forall 1 \leq j \leq i$). Given a k-dimensional PN $\mathcal{P} = (P, T, \varphi)$, suppose $p = w_1 w_2 \cdots w_l$ is a sequence of vectors (generalized markings) in $\mathbb{Z}^k$ such that $\forall j, 1 \leq j < l$, $z_{j+1} - z_j \in \bar{T}$ (the set of transition vectors). Sequence p is said to be

- *i-bounded* (resp., *i-r bounded*) if every member of p is *i-bounded* (resp., *i-r bounded*).
- *i-covering* with respect to v if $w_l(j) \geq v(j)$ $\forall 1 \leq j \leq i$
- *self-covering* (resp., *$\emptyset$-self-covering*) if there is a j, $1 \leq j \leq l$ such that $w_j < w_l$ (resp. $w_j \leq w_l$).
- an *i-loop* if $w_l(j) = w_1(j), \forall 1 \leq j \leq i$.

First consider the set *NOTBLOCKED($\hat{T}$)*. It is reasonably easy to see that a vector $\mu \in$ *NOTBLOCKED($\hat{T}$)* iff there exists a $t \in \hat{T}$, and a $\mu' \in R(\mathcal{P}, \mu)$ such that $\mu' \xrightarrow{t}$. Notice that $\mu' \xrightarrow{t}$ is equivalent to $\forall p \in P, \mu'(p) \geq \varphi(p, t)$, which is exactly the problem of asking whether there exists a computation from μ to some reachable vector μ' which *covers* t^-, where $t^- \in \mathbb{N}^k$ is defined as $t^-(i) = \varphi(p_i, t)$ $\forall 1 \leq i \leq k$. The question is in fact an instance of the well-known *covering* problem[1] of PNs. As a result, *NOTBLOCKED($\hat{T}$)*$=\bigcup_{t \in \hat{T}}$*NOTBLOCKED($\{t\}$)*$=$ $\bigcup_{t \in \hat{T}}$ $\{\mu \mid t^-$ can be covered in PN $(\mathcal{P}, \mu)\}$. To bound the size of *NOTBLOCKED($\hat{T}$)*, it suffices to find a bound for the minimal elements of each of the constituent *NOTBLOCKED($\{t\}$)*.

[1] The covering problem is that of, given a PN $(\mathcal{P}, \mu_0)$ and a marking μ, deciding whether there is a reachable marking μ' such that $\mu' \geq \mu$ (i.e., μ' covers μ).

Lemma 1. *Given a k-dimensional PN (P, T, φ), a transition $t \in T$, a $z \in \mathbb{N}_\omega^k$, $reg(z) \cap$ NOTBLOCKED$(\{t\}) \neq \emptyset$ iff there is a witness z' with $||z'|| \leq n^{2^{d \times k \times logk}}$, where $n = ||\bar{T}||$ and d is a constant. (Note that the bound is independent of z).*

Proof. (Sketch) The proof tailors the inductive approach used in [11] for the covering problem of PNs to incorporating a multi-parameter analysis.

As was done in [11], for each $w \in \mathbb{Z}^k$, let $s(i, w)$ be the length of the shortest i-bounded i-covering path (starting from w) with respect to t, if at least one such path exists; otherwise, let $s(i, w) = 0$. We define $h(i) = max\{s(i, w) \mid w \in \mathbb{N}^k\}$. It is important to note, as pointed out in [11], that $h(i)$ depends on t and $\bar{T}$, and is independent of the starting vector from which the path of the PN begins. Let n be $||\bar{T}||$, which is an upper bound on the number of tokens that can be removed from a place as a result of firing a single PN transition. Lemmas 3.3 and 3.4 of [11] show that

$$h(0) = 1 \text{ and } h(i + 1) \leq (n \times h(i))^{i+1} + h(i), \text{ for } 1 \leq i < k.$$

In order to be self-contained, a proof sketch of the above is given below. $h(0) = 1$ is trivial (as indicated in Lemma 3.3 of [11]). The key in Rackoff's proof relies on finding a bound for $h(i)$ inductively, where $i = 1, 2, ..., k$.

Let $p : v_1 \cdots v_m$ be any $(i + 1)$-covering path. Consider two cases:

- Case 1: Path p is $(i+1) - (n \times h(i))$-bounded. Then there exists an $(i+1) - (n \times h(i))$-bounded $(i+1)$-covering path with no two vectors agree on all the first $i + 1$ coordinates. Hence, the length of such a path is $\leq (n \times h(i))^{i+1}$.
- Case 2: Otherwise, let v_h be the first vector along p that is not $(n \times h(i))$ bounded. By chopping off $(i + 1)$-loops, the prefix $v_1...v_h$ can be shortened if necessary (as in Case 1) to make the length $\leq (n \times h(i))^{i+1}$. With no loss of generality, we assume the $(i + 1)$st position to be the coordinate whose value exceeds $n \times h(i)$ at v_h. Recalling the definition of $h(i)$, there is an i-covering path, say l, of length $\leq h(i)$ from v_h. By appending l to $v_1...v_h$ (i.e., replacing the original suffix path $v_h...v_m$ by l), the new path is an $(i + 1)$-bounded $(i + 1)$-covering path, because the value of the $(i + 1)$st coordinate exceeds $n \times h(i)$ and the path l (of length bounded by $\leq h(i)$) can at most subtract $(n \times h(i))$ from coordinate $i + 1$. (Note that the application of a PN transition can subtract at most n from a given coordinate.) Note that the length of the new path is bounded by $(n \times h(i))^{i+1} + h(i)$.

By solving the recurrence relation $h(0) = 1$ and $h(i+1) \leq (n \times h(i))^{i+1} + h(i)$, for $1 \leq i < k$, we have $h(k) \leq n^{2^{c \times k \times logk}}$, for some constant c. What this bound means is that regardless of the initial vector, if v can be covered then there is a covering path of length bounded by $n^{2^{c \times k \times logk}}$. Since a path of length $\leq n^{2^{c \times k \times logk}}$ can at most subtract $||\bar{T}|| \times n^{2^{c \times k \times logk}}$ from any component, the witness $||z'||$ is therefore bounded by $(||\bar{t}|| + ||\bar{T}|| \times n^{2^{c \times k \times logk}}) \leq n^{2^{d \times k \times logk}}$, for some constant d. ∎

Now consider the set *UNBOUNDED*.

Lemma 2. *Given a k-dimensional PN (P, T, φ), a $z \in \mathbb{N}^k_\omega$, $reg(z) \cap$ UNBOUNDED $\neq \emptyset$ iff there is a witness z' with $||z'|| \leq n^{2^{d\times k\times logk}}$, where $n = ||\bar{T}||$ and d is a constant. (Note that the bound is independent of z).*

Proof. (Sketch) We follow some of the results of a multi-parameter analysis of the boundedness problem for Petri nets in [12], which can be regarded as a refinement of the respective work in [11].

For each $w \in \mathbb{Z}^k$, let $m(i, w)$ be the length of the shortest i-bounded self-covering path (starting from w), if at least one such path exists; otherwise, let $m(i, w) = 0$. We define $g(i) = max\{m(i, w) \mid w \in \mathbb{Z}^k\}$. Note that $g(i)$ is independent of the initial vector of the PN. Let n be $||\bar{T}||$. Lemmas 2.3 and 2.4 of [12] show that

$$g(0) \leq n^{ck}, \text{ and } g(i+1) \leq (n^2 \times g(i))^{k^c}, \text{ for } 1 \leq i < k \text{ and some constant } c.$$

By recursively applying the above, $g(k) \leq n^{2^{c'\times k\times logk}}$, for some constant c'. The derivation of the above recurrence relation is similar to that given in the proof of our previous lemma. What this bound means is that regardless of the initial vector, the PN is unbounded iff there is a self-covering path of length bounded by $n^{2^{c'\times k\times logk}}$. Since a path of length $\leq n^{2^{c'\times k\times logk}}$ can at most subtract $n \times n^{2^{c'\times k\times logk}}$ from any place, $||z'||$ is therefore bounded by $(n \times n^{2^{c'\times k\times logk}}) \leq n^{2^{d\times k\times logk}}$, for some constant d. ∎

We now turn our attention to the set $NOTDEAD$. It is easy to see that $NOTDEAD = \{\mu \mid \mu \xrightarrow{\delta} \mu' \xrightarrow{\sigma} \mu'', \mu' \leq \mu'',\ \text{ for some } \delta \in T^*, \sigma \in T^+\}$. By noting that the condition '$\mu \xrightarrow{\delta} \mu' \xrightarrow{\sigma} \mu'', \mu' \leq \mu''$' is equivalent to having a k-bounded $\emptyset$-self-covering path from μ, an argument very similar to the UNBOUNDED case is sufficient to show the following:

Lemma 3. *Given a k-dimensional PN (P, T, φ), a $z \in \mathbb{N}^k_\omega$, $reg(z) \cap$ NOTDEAD $\neq \emptyset$ iff there is a witness z' with $||z'|| \leq n^{2^{d\times k\times logk}}$, where $n = ||\bar{T}||$ and d is a constant.*

For the set $CONTINUAL(\hat{T})$, we have the following:

Lemma 4. *Given a k-dimensional PN (P, T, φ), a subset $\hat{T} \subseteq T$, a $z \in \mathbb{N}^k_\omega$, $reg(z) \cap$ CONTINUAL($\hat{T}$) $\neq \emptyset$ iff there is a witness z' with $||z'|| \leq n^{2^{d\times k\times logk}}$, where $n = ||\bar{T}||$ and d is a constant.*

Proof. (Sketch) The proof is in principle similar to that of Lemma 2. The thing that has to be taken care of is to ensure that the $\emptyset$-self-covering path uses all the transitions in $\hat{T}$. We define a path $z_1, z_2, ..., z_p$ to be $\hat{T}$-self-covering if there exists a $1 \leq j < p$ such that $z_j \leq z_p$ and all the transitions in $\hat{T}$ are utilized in the sub-sequence $z_j, ..., z_p$. For each $w \in \mathbb{Z}^k$, let $m'(i, w)$

be the length of the shortest i-bounded $\hat{T}$-self-covering path (starting from w), if at least one such path exists; otherwise, let $m'(i,w) = 0$. We define $g'(i) = max\{m'(i,w) \mid w \in \mathbb{Z}^k\}$. Then following a very similar inductive argument as presented in [11,12], a recurrence relation $g'(0) \leq n^{ck}$, and $g'(i+1) \leq (n^2 \times g'(i))^{k^c}$, for $1 \leq i < k$ and some constant c, can be derived. The rest is easy. ∎

Theorem 3. *Given a k-dimensional PN (P, T, φ) and a $\hat{T} \subseteq T$, $||min($NOTBLOCKED$(\hat{T}))||$, $||min($UNBOUNDED$)||$, $||min($NOTDEAD$)||$, $||min($CONTINUAL$(\hat{T}))|| \leq n^{2^{d \times k \times logk}}$, where $n = ||\bar{T}||$ and d is a constant.*

Proof. (Sketch) Given a $z \in \mathbb{N}_\omega^k$, define $f(||z||) = n^{2^{d \times k \times logk}}$ (where $n = ||\bar{T}||$ and d a constant stated in Lemma 1) which provides an upper bound for a witness certifying $reg(z) \cap$ NOTBLOCKED$(\hat{T}) \neq \emptyset$, if one exists. Notice that the value of f is independent of z. Our result follows immediately from Theorem 2. From Lemmas 2, 3, and 4, the other three cases are similar. ∎

The results in Theorem 3 can easily be modified for the model of VASSs. It is interesting to point out that the minimal elements of a number of upward-closed sets concerning infinite computations of Petri nets with respect to various notions of *fairness* (see Definition 6.9 of [13]) can also be obtained from the result of Theorem 3, as the set *CONTINUAL* is sufficient to capture such infinite fair computations.

Now we consider a problem that arises frequently in *automated verification*. Given a system S with initial state q, and a designated set of states Q, it is often of interest and importance to ask whether some state in Q can be reached from q, which constitutes a question related to the analysis of a *safety property*. Instead of using the *forward-reachability analysis* (which computes all the states that can be reached from q to see whether the intersection with Q is non-empty or not), an equally useful approach is to use the so-called *backward-reachability analysis*. In the latter, we compute the set $pre^*(S, Q)$ which consists of all the states from which some state in Q is reachable, and then decide whether $q \in pre^*(S, Q)$. In general, $pre^*(S, Q)$ may not be computable for infinite state systems.

For PNs, we define the *backward-reachability* (*BR*, for short) problem as follows:

- **Input:** A PN $\mathcal{P}$ and a set U of markings
- **Output:** The set $pre^*(\mathcal{P}, U) = \{\mu \mid R(\mathcal{P}, \mu) \cap U \neq \emptyset\}$

In words, the problem is to find the set of initial markings from which a marking in U can be reached. Now suppose U is upward-closed, then $\{\mu \mid R(\mathcal{P}, \mu) \cap U \neq \emptyset\}$ is upward-closed as well, and is, in fact, equivalent to $\bigcup_{\nu \in min(U)} \{\mu \mid \exists \mu' \in R(\mathcal{P}, \mu), \mu' \geq \nu\}$. The latter is basically asking about coverability issues of PNs. Hence, the max-value of the minimal elements can be derived along the same line as that for the set *NOTBLOCKED*.

4.2 Parametric Clocked Petri Nets

Clocked Petri nets are Petri nets augmented by a finite set of real-value *clocks* and *clock constraints*. Clocks are used to measure the progress of real-time in the system. All the clocks are resetable and increase at a uniform rate. We can as well regard clocks as stop-watches which refer to the same global clock. The use of such clock structure was originally introduced in [2] for defining *timed automata*.

Given a set $X = \{x_1, x_2, \ldots, x_h\}$ of clock variables, the set $\Phi(X)$ of clock constraints δ is defined inductively by

$$\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta \wedge \delta,$$

where x is a clock in X and c is a constant in $\mathbb{Q}^+$ (i.e., the set of nonnegative rationals). A *clock reading* is a mapping $\nu : X \to \mathbb{R}$ which assigns each clock a real value. For $\eta \in \mathbb{R}$, we write $\nu + \eta$ to denote the clock reading which maps every clock x to the value $\nu(x) + \eta$. That is, after η time units added to the global clock, the value of every clock must increase by η units as well. A clock reading ν for X *satisfies* a clock constraint δ over X, denoted by $\delta(\nu) \equiv$ **true** , iff δ evaluates to **true** using the values given by ν.

A *clocked Petri net* is a 6-tuple $\mathcal{N} = (P, T, \varphi, X, r, q)$, where (P, T, φ) is a PN, X is a finite set of real-value clock variables, $r : T \longrightarrow 2^X$ is a labeling function assigning clocks to transitions, and $q : T \longrightarrow \Phi(X)$ is a labeling function assigning clock constraints to transitions. Intuitively, $r(t)$ contains those clock variables which are reset when transition t is fired. A *configuration* (μ, η, ν) of a clocked Petri net consists of a *marking* μ, the *global time* η and the present *clock reading* ν. Note that the clock reading ν is continuously being updated as η, the global time, advances. Hence, ν and η are not completely independent. Given a configuration (μ, η, ν) of a clocked Petri net $\mathcal{P}$, a transition t is *enabled* iff $\forall\, p \in P,\ \varphi(p, t) \leq \mu(p)$, and ν satisfies $q(t)$, the set of constraints associated with transition t, i.e., $q(t)(\nu) \equiv$ **true** . Let μ be the marking and ν the clock reading at time η. Then t *may* fire at η if t is enabled in the marking μ with the clock reading ν. We then write $(\mu, \nu) \overset{(t,\eta)}{\to} (\mu', \nu')$, where $\mu'(p) = \mu(p) - \varphi(p, t) + \varphi(t, p)$ (for all $p \in P$), and $\nu'(x) = 0$ (for all $x \in r(t)$). Note that the global time remains unchanged as a result of firing t. That is, the firing of a transition is assumed to let no time elapse at all. The global clock will start moving immediately after the firing of a transition is completed. Initially, we assume the initial global time η_0 and clock reading ν_0 to be $\eta_0 = 0$ and $\nu_0(x) = 0 (\forall x \in X)$, respectively. It is important to point out that, as opposed to timed PNs under *urgent firing semantics*, *enabledness* is necessary but not sufficient for transition firing in a clocked PN. In other words, it is <u>not</u> required to fire all the enabled transitions at any point in time during the course of the computation.

Now consider clocked PNs with parameterized constraints. That is, the 'c' in the atomic constraints $x \leq c$ and $c \leq x$ is not a constant; instead, it is an unknown *parameter*. We are interested in the following question:

- *Input*: Given a clocked PN $\mathcal{P}$ with unknown integer parameters $\theta_1, \cdots, \theta_n$ in its clock constraints, and a set Q of goal markings

- *Output*: find the values of $\theta_1, \cdots, \theta_n$ (if they exist) so that there exists a computation reaching a marking in Q. In what follows, we let $S(\theta_1, \cdots, \theta_n)$ denote such a set of solutions.

Even for timed automata, it is known that the emptiness problem (i.e., the problem of deciding whether there exists a parameter setting under which the associated timed language is empty or not) is undecidable when three or more clocks are compared with unknown parameters [3]. In what follows, we consider a special case in which the involved atomic clock constraints are only of the form $x \leq \theta$ or $x < \theta$, and there are no negative signs immediately before inequalities. In this case, the set '$\{(\theta_1, \cdots, \theta_n) \mid$ there exists a computation in $\mathcal{P}$ reaching a marking in Q under $(\theta_1, \cdots, \theta_n)\}$' is clearly upward-closed, as $x \leq \theta \Longrightarrow x \leq \theta'$ and $x < \theta \Longrightarrow x < \theta'$, if $\theta \leq \theta'$. That is, whatever enabled under θ is also enabled under θ'.

A technique known to be useful for reasoning about timed automata is based on the notion of 'equivalence classes' [2]. In spite of the differences in value, two different clock readings may induce identical system's behaviors; in this case, they are said to be in the same clock region. For clock constraints falling into the types given in our setting, the number of distinct clock regions is always finite [2], meaning that a timed automaton (which is of infinite-state potentially) is equivalent behaviorally to a so-called *region automaton* (which is of finite-state). Furthermore, the number of clock regions of a timed automaton A is bounded by $2|Q|(|X| \cdot (C_A + 2))^{|X|}$, where $|Q|$ is the number of states of A, $|X|$ is the number of clocks of A, and C_A is the maximum timing constant involved in A (see [2] for details). This, coupled with our earlier discussion of upward-closed sets and PNs, yields the following result:

Theorem 4. *Given a k-dimensional clocked PN $\mathcal{P}$ with unknown integer parameters $\theta_1, \cdots, \theta_n$ in its clock constraints, and an upward-closed set Q of goal markings, $||min(S(\theta_1, \cdots, \theta_n))||$ is bounded by $O((D \cdot |X|)^{2^{d_2 \cdot n \cdot k \cdot logk} \cdot |X|^n})$, where $|X|$ is the number of clocks, D is the absolute value of the maximum number involved in $\mathcal{P}$ and $min(Q)$, and d_2 is a constant.*

Proof. (Sketch) The proof is somewhat involved, and hence, only a proof sketch is given here. Our derivation is based on the approach detailed in Theorem 2.

For the PN to reach Q, it suffices to consider whether a marking covering an element in $min(Q)$ is reachable or not. Recall from Theorem 2 that our approach for computing $||min(S(\theta_1, \cdots, \theta_n))||$ begins by letting $(\theta_1, \cdots, \theta_n)$=$(\omega, \cdots, \omega) = v_0$. In this case, the associated clocked PN can be simplified by deleting all the clock constraints involving θ_i, because $x \leq (<) \omega$ always holds. Now the idea is to simulate clocked PNs by VASSs. To this end, we use the 'state' portion of the VASS to capture the structure of (finitely many) clock regions of a clocked PN as discussed earlier, and an 'addition vector' of the VASS to simulate a transition of the PN. Using the analysis of [2], it is not hard to see that the number of clock regions is bounded by $O((|X| \cdot C_0)^{|X|})$, where $|X|$ is the number of clocks and C_0 is the maximum timing constant (i.e., maximum value of constants involved in clock constraints), which corresponds to the number of states of the VASS.

It was shown in [12], using the technique of multi-parameter analysis, that for an m-state, k-dimensional VASS whose largest integer can be represented in l bits, the length of the shortest witnessing path covering a given marking is bounded by $O((2^l \cdot m)^{2^{d \cdot k \cdot logk}})$. Applying a similar analysis to our constructed VASS and the concept of clock regions, a witness for '$reg(v_0) \cap S(\theta_1, \cdots, \theta_n)) \neq \emptyset$?' (if it exists) of max-value bounded by $d_1 \cdot (D \cdot (|X| \cdot C_0)^{|X|})^{2^{d_2 \cdot k \cdot logk}}$ can be found, for some constants d_1, d_2. This bound corresponds to the b value in the statement of Theorem 2.

The next step in to start with $v_1 = (\theta_1, \omega, ..., \omega)$ with $\theta_1 < d_1 \cdot (D \cdot (|X| \cdot C_0)^{|X|})^{2^{d_2 \cdot k \cdot logk}}$. Let this value be C_1. In this case, those clock constraints involving θ_1 can no longer be ignored. We construct a new VASS simulating the associated clocked PN, and such a VASS has its number of states bounded by $O((|X| \cdot C_1)^{|X|})$, implying that a witness for '$reg(v_1) \cap S(\theta_1, \cdots, \theta_n)) \neq \emptyset$?' (if it exists) of max-value bounded by $d_1 \cdot (D \cdot (|X| \cdot C_1)^{|X|})^{2^{d_2 \cdot k \cdot logk}}$ can be found, which corresponds to the f function (w.r.t. variable C_1) in Theorem 2. Finally, Theorem 2 immediately yields $||min(S(\theta_1, \cdots, \theta_n))|| = O((D \cdot |X|)^{2^{d_2 \cdot n \cdot k \cdot logk} \cdot |X|^n})$. ∎

4.3 Subclasses of Petri Nets

In this section, we show how Theorem 2 can be applied to deriving bounds for upward-closed sets associated with a variety of subclasses of PNs depicted in Figure 2. The problem under consideration is as follows:

- **Input:** A PN $\mathcal{P} = (P, T, \varphi)$, a marking μ', and a system of linear (in)equalities $L(v_1, ..., v_m)$, where $m = |T|$. Clearly, the set $pre^*(\mathcal{P}, (\mu', L)) = \{\mu \mid \exists \sigma \in T^*,\ \mu \xrightarrow{\sigma} \mu'',\ \mu'' \geq \mu'$ and $L(\#_\sigma(t_1), \cdots, \#_\sigma(t_m))$ holds $\}$ is upward-closed. (Intuitively, the set contains those markings μ from which there is a computation covering μ' and along which the transition firing count vector $(\#_\sigma(t_1), \cdots, \#_\sigma(t_m))$ satisfies L.)
- **Output:** $min(pre^*(\mathcal{P}, (\mu', L)))$

What makes the subclasses of PNs in Figure 2 of interest is that their reachability sets can be characterized by *integer linear programming* (*ILP*) – a relatively well-understood mathematical model (see [16]). In our subsequent discussion, we shall use normal PNs as an example to show how to derive the max-value of the minimal elements of the pre^* associated with a normal PN and an upward-closed goal set U. Before going into details, the definition of *normal PNs* is in order.

A *circuit* of a PN is a 'simple' closed path in the PN graph. (By 'simple' we mean all nodes are distinct along the closed path.) Given a PN $\mathcal{P}$, let $c = p_1 t_1 p_2 t_2 \cdots p_n t_n p_1$ be a circuit and let μ be a marking. Let $P_c = \{p_1, p_2, \cdots, p_n\}$ denote the set of places in c. We define the *token count* of circuit c in marking μ to be $\mu(c) = \sum_{p \in P_c} \mu(p)$. A PN is *normal* [15] iff for every minimal circuit c and transition t_j, $\sum_{p_i \in P_c} a_{i,j} \geq 0$, where $a_{i,j} = \varphi(t_j, p_i) - \varphi(p_i, t_j)$. Hence, for every minimal circuit c and transition t in a normal PN, if one of t's input places is in

c, then one of t's output places must be in c as well. Intuitively, a PN is normal iff no transition can decrease the token count of a minimal circuit by firing at any marking. For the definitions and the related properties for the rest of the PNs in Figure 2, the reader is referred to [16].

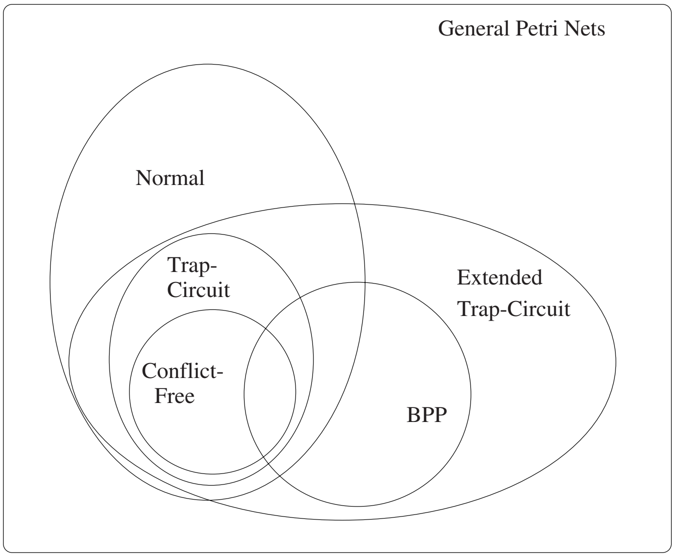


Fig. 2. Containment relationship among subclasses of PNs

In [10], the reachabiliy problem of normal PNs was equated with ILP using the so-called decompositional approach. The idea behind the decompositional technique relies on the ability to decompose a PN $\mathcal{P}$=(P, T, φ) (possibly in a nondeterministic fashion) into sub-PNs $\mathcal{P}_i = (P, T_i, \varphi_i)$ $(1 \leq i \leq n$, $T_i \subseteq T$, and φ_i is the restriction of φ to $(P \times T_i) \cup (T_i \times P))$ such that for an arbitrary computation $\mu_0 \xrightarrow{\sigma} \mu$ of PN $\mathcal{P}$, σ can be rearranged into a canonical form $\sigma_1\sigma_2 \cdots \sigma_n$ with $\mu_0 \xrightarrow{\sigma_1} \mu_1 \xrightarrow{\sigma_2} \mu_2 \cdots \mu_{n-1} \xrightarrow{\sigma_n} \mu_n = \mu$, and for each i, a system of linear inequalities $ILP_i(x, y, z)$ can be set up (based upon sub-PN $\mathcal{P}_i$, where x, y, z are vector variables) in such a way that $ILP_i(\mu_{i-1}, \mu_i, z)$ has a solution for z iff there exists a σ_i in T_i^* such that $\mu_{i-1} \xrightarrow{\sigma_i} \mu_i$ and $z = \#_{\sigma_i}$. See Figure 3.

Consider a normal PN $\mathcal{P} = (P, T, \varphi)$ and let $P = \{p_1, ..., p_k\}$ and $T = \{t_1, ..., t_m\}$. In [10], it was shown that an arbitrary computation of a normal PN can be decomposed according to a sequence of distinct transitions $\tau = t_{j_1} \cdots t_{j_n}$. More precisely, we define the *characteristic system of inequalities* for $\mathcal{P}$ and τ as $S(\mathcal{P}, \tau) = \bigcup_{1 \leq h \leq n} S_h$, where

- $S_h = \{x_{h-1}(i) \geq \varphi(p_i, t_{j_h}), x_h = x_{h-1} + A_h \cdot y_h \mid 1 \leq i \leq k\}$, A_h is an $k \times h$ matrix whose columns are $\bar{t}_{j_1}, \cdots, \bar{t}_{j_h}$, y_h is a $h \times 1$ column vector, for $1 \leq h \leq n$.

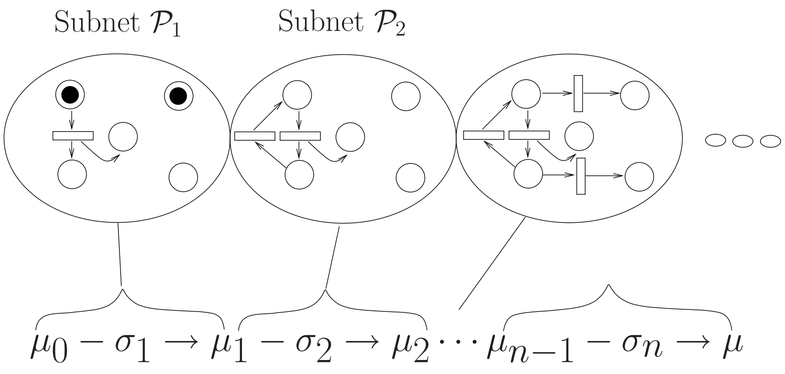

Subnet $\mathcal{P}_1$
Subnet $\mathcal{P}_2$
$\mu_0 - \sigma_1 \rightarrow \mu_1 - \sigma_2 \rightarrow \mu_2 \cdots \mu_{n-1} - \sigma_n \rightarrow \mu$

Fig. 3. Decompositional approach

The variables in S are the components of the k-dimensional column vectors $x_0, ..., x_n$ and the h-dimensional column vectors $y_h, 1 \leq h \leq n$. In [10], it was shown that $\mu' \in R(\mathcal{P}, \mu)$ iff there exists a sequence of distinct transitions $\tau = t_{j_1} \cdots t_{j_n}$ such that $\{x_0 = \mu\} \cup \{x_n = \mu'\} \cup S(\mathcal{P}, \tau)$ has a nonnegative integer solution. In particular, for each $1 \leq h \leq n$, the i-th coordinate of the y_h variable (an $h \times 1$ column vector) represents the number of times transition t_{j_i} $(1 \leq i \leq h)$ is used along the path reaching from x_{h-1} to x_h. Intuitively speaking, the decomposition is carried out in such a way that

– stage h involves one more transition (namely t_{j_h}) than its preceding stage $h-1$; furthermore, t_{j_h} must be enabled in x_{h-1} as the condition '$x_{h-1}(i) \geq \varphi(p_i, t_{j_h})$' in S_h enforces,
– x_h represents the marking at the end of stage h and the begining of stage $h+1$,
– the computation from x_{h-1} to x_h is captured by S_h, in which '$x_h = x_{h-1} + A_h \cdot y_h$' simply says that the *state equation* associated with the sub-PN in stage h is sufficient and necessary to capture reachability between two markings.

For convenience, we define y'_h to be a vector in $\mathbb{N}^m$ such that $y'_h(j_i) = y_h(i)$, $1 \leq i \leq h$, and the remaining coordinates are zero. Note that y'_h is an m dimensional vector w.r.t. the ordering $t_1, t_2, ..., t_m$ and y_h is an h dimensional vector w.r.t. the ordering $t_{j_1}, t_{j_2}, ..., t_{j_h}$. Intuitively speaking, $y'_h(i)$ serves as the purpose of rearranging the vector y_h w.r.t. the ordering $t_1, t_2, ..., t_m$, while filling those coordinates not corresponding to $t_{j_1}, t_{j_2}, ..., t_{j_h}$ with zero.

Before deriving our result, we need the following concerning the size of the solutions of ILP instances.

Lemma 5. *(From [5]) Let $d_1, d_2 \in \mathbb{N}^+$, let B be a $d_1 \times d_2$ integer matrix and let b be a $d_1 \times 1$ integer matrix. Let $d \geq d_2$ be an upper bound on the absolute values the integers in B and b. If there exists a vector $v \in \mathbb{N}^{d_2}$ which is a solution to*

$Bv \geq b$, then for some constant c independent of d, d_1, d_2, there exists a vector v such that $Bv \geq b$ and $||v|| \leq d^{cd_1}$.

Now we are in a position to derive a bound for the minimal elements of pre^* for normal PNs.

Theorem 5. *Given a normal PN $\mathcal{P} = (P, T, \varphi)$ with $|P| = k$ and $|T| = m$, a marking μ', and a linear constraint $L(v_1, ..., v_m)$, then $||min(pre^*(\mathcal{P}, (\mu', L)))|| \leq (a_1)^{(c*a_2)^k}$, where c is some constant, $a_1 = max\{||\bar{T}||, s, (m+k)*m\} * m * ||\bar{T}||$, $a_2 = (m * k + r)$, r the number of (in)equalities in L, and s the absolute value of the largest integer mentioned in L.*

Proof. Given a subset of places $Q \subseteq P$, we define a *restriction* of $\mathcal{P}$ on Q as PN $\mathcal{P}_Q = (Q, T, \varphi_Q)$, where φ_Q is the restriction of φ on Q and T (i.e., $\varphi_Q(p,t) = \varphi(p,t); \varphi_Q(t,p) = \varphi(t,p)$ if $p \in Q$). It is obvious from the definition of normal PNs that $\mathcal{P}_Q$ is normal as well.

Now consider a vector $v \in \mathbb{N}_\omega^k$. To find a witness for $reg(v) \cap pre^*(\mathcal{P}, (\mu', L)) \neq \emptyset$, if one exists, it suffices to consider sub-PN with $Q(v) = \{p \mid v(p) \neq \omega,\ p \in P\}$ as the set of places (as opposed to the original set P), since each ω place can supply an arbitrary number of tokens to each of its output transitions. (That is, places associated with ω components in v can be ignored as far as reaching a goal marking in U is concerned.) Hence, $reg(v) \cap pre^*(\mathcal{P}, (\mu', L)) \neq \emptyset$ iff for some τ (of length $\leq m$), the following system of linear inequalities has a solution

$$H \equiv S(\mathcal{P}_Q, \tau) \cup \{x_0 = v\} \cup \{x_n \geq \mu'\} \cup L(v_1, ..., v_m) \cup \{(v_1, ..., v_m)^{tr} = y'_1 + ... + y'_n\}.$$[2]

Notice that in the above, $\{(v_1, ..., v_m)^{tr} = y'_1 + ... + y'_n\}$ ensures that for each transition t_i, the number of times t_i being used in the computation (i.e., $y'_1(i) + y'_2(i) + \cdots + y'_n(i)$) captured by $S(\mathcal{P}_Q, \tau)$ equals v_i. Recall that y'_h captures the firing count vector in segment h.

A careful examination reveals that in H, the number of inequalities is bounded by $O(m * k + r)$, and the number of variables is bounded by $O((m + k) * m)$. Furthermore, the absolute value of the maximal numbers in H is bounded by $max\{||v||, ||\bar{T}||, s\}$. Using Lemma 5, if H has a solution, then a 'small' solution of max-value bounded by $(max\{||v||, ||\bar{T}||, s, (m+k)*m\})^{b*(m*k+r)}$ exists, for some constant b. Recall that the y'_h vector variable represents the numbers of times the respective transitions are used along segment h of the reachability path. As a result, an initial marking with at most $m * ((max\{||v||, ||\bar{T}||, s, (m + k) * m\})^{b*(m*k+r)}) * ||\bar{T}||$ tokens in each of the ω places suffices for such a path to be valid in the original PN, since each transition consumes at most $||\bar{T}||$ tokens from a place. The above is bounded by $(a_1 * ||v||)^{b*a_2}$ (for a_1, a_2 given in the statement of the theorem), where b is a constant. Now define $f(||v||) = (a_1 * ||v||)^{b*a_2}$. From Theorem 2, $||min(pre^*(\mathcal{P}, (\mu', L)))||$ is bounded by $f^{(k-1)}(||(\omega, ..., \omega)||)$, which can easily be shown to be bounded by $(a_1)^{(c*a_2)^k}$, for some constant c. ∎

[2] The superscript *tr* denotes the transpose of a matrix.

The above theorem provides a framework for analyzing a number of upward-closed sets associated with normal PNs. The BR problem mentioned at the end of Section 4.1 clearly falls into this category. Our results for normal PNs carry over to the rest of the subclasses listed in Figure 2, although the bounds are slightly different. Due to space limitations, the details are omitted here.

4.4 Lossy Petri Nets

Lossy Petri nets (or equivalently, *lossy vector addition systems with states*) were first defined and investigated in [4] with respect to various model checking problems. A *lossy Petri net* (P, T, φ) is a PN for which tokens may be *lost* spontaneously without transition firing during the course of a computation. To be more precise, an *execution step* from markings μ to μ', denoted by $\mu \Rightarrow \mu'$, of a lossy PN can be of one of the following forms: (1) $\exists t \in T, \mu \xrightarrow{t} \mu'$, or (2) $\mu > \mu'$ (denoting token loss spontaneously at μ). As usual, $\overset{*}{\Rightarrow}$ is reflexive and transitive closure of $\Rightarrow$. It is easy to observe that for arbitrary goal set U (not necessarily upward-closed), the set $pre^*(\mathcal{P}, U)$ for lossy PN $\mathcal{P}$ is always upward-closed. Consider the case when $U = \{\mu'\}$. We have:

Theorem 6. *Given a lossy PN $\mathcal{P} = (P, T, \varphi)$, and a goal marking μ', the minimal elements of the set $pre^*(\mathcal{P}, \{\mu'\}) = \{\mu \mid \mu \overset{*}{\Rightarrow} \mu'\}$ have their max-values bounded by $n^{2^{d \times k \times log k}}$, where $n = max\{||\bar{T}||, ||\mu'||\}$ and d is a constant.*

Proof. It is not hard to see that $pre^*(\mathcal{P}, \{\mu'\}) = \{\mu \mid \mu \overset{*}{\Rightarrow} \mu'\} = \{\mu \mid \mu \xrightarrow{*} \mu'', \mu'' \geq \mu'\}$, which involves coverability-related queries. Hence, the result follows from our discussion in Section 4.1. ∎

The above result can be easily extended to the case when U is an upward-closed set.

In [4], the *global model checking* problem for (conventional or lossy) VASSs with respect to formula of the form $\exists \mathcal{A}_\omega(\pi_1, ..., \pi_m)$ has been shown to be decidable. We shall see how our unified strategy as well as results obtained previously can be applied to deriving complexity bound for the above model checking problem. Before doing that, some definitions, many of which can be found in [4], are needed.

An *upward-closed constraint* π over variable set $X = \{x_1, ..., x_k\}$ is of the form $\bigvee_{x_i \in X} x_i \geq c_i$, where $c_i \in \mathbb{N}, 1 \leq i \leq k$. A k-dimensional vector v is said to satisfy π, denoted by $v \models \pi$, if $v(i) \geq c_i$, $\forall 1 \leq i \leq k$.

Consider a k-dim VASS $\mathcal{V} = (v_0, V, s_1, S, \delta)$ with $S = \{s_1, ..., s_h\}$. Given h upward-closed constraints $\pi_1, ..., \pi_h$ over variable set $X = \{x_1, ..., x_k\}$, and a configuration σ_1, we write $\sigma_1 \models \exists_\omega(\pi_1, ..., \pi_h)$, iff there is an infinite computation $\sigma_1, \sigma_2, \cdots, \sigma_i, \cdots$, such that $\forall i \geq 1$, if $state(\sigma_i) = s_j$, then $val(\sigma_i) \models \pi_j$. In words, there exists an infinite path from configuration σ_1 along which the vector value of each configuration satisfies the upward-closed constraint associated with the state of the configuration. In [4], it was shown the following *global model checking problem* to be decidable:

- Given a k-dim VASS $\mathcal{V}= (v_0, V, s_1, S, \delta)$ with $S = \{s_1, ..., s_h\}$ and a formula $\phi = \exists_\omega(\pi_1, ..., \pi_h)$, for upward-closed constraints $\pi_1, ..., \pi_h$,
- Output: The set $[[\phi]]_{\mathcal{V}}=\{\sigma \mid \sigma \models \phi \text{ in } \mathcal{V}\}$.

The following result gives a complexity bound for the above problem.

Theorem 7. *For each state $s \in S$, $||min(\{v \in \mathbb{N}^k \mid (s, v) \in [[\phi]]_{\mathcal{V}}\})||$ is bounded by $n^{2^{d\times k\times logk}}$, where $n = max\{||\bar{T}||, u\}$, u is the absolute value of the largest number mentioned in ϕ, and d is a constant.*

Proof. (Sketch) The proof is done by constructing a VASS $\mathcal{V}'=(v'_0, V', s'_1, S', \delta')$ from $\mathcal{V}$ such that $(s_1, v_0) \models \phi$ in $\mathcal{V}$ iff there exists an infinite path from (s'_1, v'_0) in $\mathcal{V}'$.

Assume that $\pi_i = \bigvee_{1\leq l\leq k}(x_l \geq c_{i,l})$. For convenience, for a value c and an index l, we define $[c]_l$ to be a vector whose $l-th$ coordinate equals c; the rest of the coordinates are zero. The construction is as follows:

- $S' = S \cup \{q_{i,l,j} \mid 1 \leq i \leq h, 1 \leq l \leq k, 1 \leq j \leq h\}$
- For each addition rule $v \in \delta(s_i, s_j)$, k addition rules are used to test the k primitive constraints in π_i, by including the following rules: $\forall 1 \leq l \leq k$, $[-c_{i,l}]_l \in \delta'(s_i, q_{i,l,j})$. Furthermore, we also have $(v + [c_{i,l}]_l) \in \delta'(q_{i,l,j}, s_j)$ to restore the testing of $c_{i,l}$ as well as adding vector v.
- $v'_0 = v_0$ and $s'_1 = s_1$

Based upon the above construction, it is reasonably easy to show that $(s_1, v_0) \models \phi$ in $\mathcal{V}$ iff there exists an infinite path from (s'_1, v'_0) in $\mathcal{V}'$. The bound of the theorem then follows from Theorem 3. ∎

5 Conclusion

We have developed a unified strategy for computing the sizes of the minimal elements of certain upward-closed sets associated with Petri nets. Our approach can be regarded as a refinement of [13] in the sense that complexity bounds become available (as opposed to merely decidability as was the case in [13]), as long as the size of a witness for a key query is known. Several upward-closed sets that arise in the theory of Petri nets as well as in backward-reachability analysis in automated verification have been derived in this paper. The reader is referred to [8] [9] for related results concerning complexity and decidability of some upward-closed sets associated with Petri nets. It would be interesting to seek additional applications of our technique.

Acknowledgments. The authors thank the anonymous referees for their comments that improved the presentation of this paper.

References

1. Abdulla, P., Cerans, K., Jonsson, B., Tsay, Y.: Algorithmic analysis of programs with well quasi-ordered domains. Information and Computation 160(1/2), 109–127 (2000)
2. Alur, R., Dill, D.: A theory of timed automata. Theoret. Comput. Sci. 126, 183–235 (1994)
3. Alur, R., Henzinger, T., Vardi, M.: Parametric real-time reasoning. In: Proc. 25th ACM STOC, pp, 592–601 (1993)
4. Bouajjani, A., Mayr, R.: Model checking lossy vector addition systems. In: Meinel, C., Tison, S. (eds.) STACS' 99. LNCS, vol. 1563, pp. 323–333. Springer, Heidelberg (1999)
5. Borosh, I., Treybis, L.: Bounds on positive integral solutions to linear diophantine equations. Proc. Amer. Math. Soc. 55, 299–304 (1976)
6. Delzanno, G., Raskin, J.: Symbolic representation of upward-closed sets. In: Schwartzbach, M.I., Graf, S. (eds.) ETAPS 2000 and TACAS 2000. LNCS, vol. 1785, pp. 426–440. Springer, Heidelberg (2000)
7. Delzanno, G., Raskin, J., Van Begin, L.: Covering sharing-trees: a compact data-structure for parametrized verification. In: J. Software Tools for Technology Transfer, vol. 5(2-3), pp. 268–297. Springer, Heidelberg (2004)
8. Esparza, J.: Decidability and complexity of Petri net problems - an introduction. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 374–428. Springer, Heidelberg (1998)
9. Esparza, J., Nielsen, M.: Decidability issues for Petri nets - a survey. Journal of Informatik Processing and Cybernetics 30(3), 143–160 (1994)
10. Howell, R., Rosier, L., Yen, H.: Normal and sinkless Petri nets. J. Comput. System Sci. 46, 1–26 (1993)
11. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoret. Comput. Sci. 6, 223–231 (1978)
12. Rosier, L., Yen, H.: A multiparameter analysis of the boundedness problem for vector addition systems. J. Comput. System Sci. 32, 105–135 (1986)
13. Valk, R., Jantzen, M.: The residue of vector sets with applications to decidability in Petri nets. Acta Informatica 21, 643–674 (1985)
14. Wang, F., Yen, H.: Timing parameter characterization of real-time systems. In: Ibarra, O.H., Dang, Z. (eds.) CIAA 2003. LNCS, vol. 2759, pp. 23–34. Springer, Heidelberg (2003)
15. Yamasaki, H.: Normal Petri nets. Theoret. Comput. Sci. 31, 307–315 (1984)
16. Yen, H.: Integer linear programming and the analysis of some Petri net problems. Theory of Computing Systems 32(4), 467–485 (1999)

ProM 4.0: Comprehensive Support for *Real* Process Analysis

W.M.P. van der Aalst[1], B.F. van Dongen[1], C.W. Günther[1], R.S. Mans[1], A.K. Alves de Medeiros[1], A. Rozinat[1], V. Rubin[2,1], M. Song[1], H.M.W. Verbeek[1], and A.J.M.M. Weijters[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
w.m.p.v.d.aalst@tue.nl
[2] University of Paderborn, Paderborn, Germany

Abstract. This tool paper describes the functionality of *ProM*. Version 4.0 of ProM has been released at the end of 2006 and this version reflects recent achievements in *process mining*. Process mining techniques attempt to extract non-trivial and useful information from so-called "event logs". One element of process mining is *control-flow discovery*, i.e., automatically constructing a process model (e.g., a Petri net) describing the causal dependencies between activities. Control-flow discovery is an interesting and practically relevant challenge for Petri-net researchers and ProM provides an excellent platform for this. For example, the theory of regions, genetic algorithms, free-choice-net properties, etc. can be exploited to derive Petri nets based on example behavior. However, as we will show in this paper, the functionality of ProM 4.0 is not limited to control-flow discovery. ProM 4.0 also allows for the discovery of other perspectives (e.g., data and resources) and supports related techniques such as conformance checking, model extension, model transformation, verification, etc. This makes ProM a versatile tool for process analysis which is not restricted to model analysis but also includes log-based analysis.

1 Introduction

The first version of ProM was released in 2004. The initial goal of ProM was to unify process mining efforts at Eindhoven University of Technology and other cooperating groups [4]. Traditionally, most analysis tools focusing on processes are restricted to *model-based analysis*, i.e., a model is used as the starting point of analysis. For example, the alternating-bit protocol can be modeled as a Petri net and verification techniques can then be used to check the correctness of the protocol while simulation can be used to estimate performance aspects. Such analysis is only useful *if the model reflects reality*. Process mining techniques use *event logs* as input, i.e., information recorded by systems ranging from information systems to embedded systems. Hence the starting point is not a model but the observed reality. Therefore, we use the phrase *real process analysis* to position process mining with respect to classical model-based analysis. Note that

ProM also uses models (e.g., Petri nets). However, these models (1) are discovered from event logs, (2) are used to reflect on the observed reality (conformance checking), or (3) are extended based on information extracted from logs.

Process mining is relevant since more and more information about processes is collected in the form of event logs. The widespread use of information systems, e.g., systems constructed using ERP, WFM, CRM, SCM, and PDM software, resulted in the omnipresence of vast amounts of event data. Events may be recorded in the form of audit trails, transactions logs, or databases and may refer to patient treatments, order processing, claims handling, trading, travel booking, etc. Moreover, recently, more and more devices started to collect data using TCP/IP, GSM, Bluetooth, and RFID technology (cf. high-end copiers, wireless sensor networks, medical systems, etc.).

Table 1. Comparing ProM 1.1 presented in [7] with ProM 4.0

Version	ProM 1.1	ProM 4.0
Mining plug-ins	6	27
Analysis plug-ins	7	35
Import plug-ins	4	16
Export plug-ins	9	28
Conversion plug-ins	3	22
Log filter plug-ins	0	14
Total number of plug-ins	29	142

At the Petri net conference in 2005, Version 1.1 of ProM was presented [7]. In the last two years ProM has been extended dramatically and currently dozens of researchers are developing plug-ins for ProM. ProM is open source and uses a plug-able architecture, e.g., people can add new process mining techniques by adding plug-ins without spending any efforts on the loading and filtering of event logs and the visualization of the resulting models. An example is the plug-in implementing the α-algorithm [5], i.e., a technique to automatically derive Petri nets from event logs. The version of ProM presented at the Petri net conference in 2005 (Version 1.1) contained only 29 plug-ins. Version 4.0 provides 142 plug-ins, i.e., there are almost five times as many plug-ins. Moreover, there have been spectacular improvements in the quality of mining algorithms and the scope of ProM has been extended considerably. This is illustrated by Table 1 which compares the version presented in [7] with the current version. To facilitate the understanding of Table 1, we briefly describe the six types of plug-ins:

- *Mining plug-ins* implement some mining algorithm, e.g., the α-miner to discover a Petri net [5] or the social network miner to discover a social network [1].
- *Export plug-ins* implement some "save as" functionality for specific objects in ProM. For example, there are plug-ins to save Petri nets, EPCs, social networks, YAWL, spreadsheets, etc. often also in different formats (PNML, CPN Tools, EPML, AML, etc.).

- *Import plug-ins* implement an "open" functionality for specific objects, e.g., load instance-EPCs from ARIS PPM or BPEL models from WebSphere.
- *Analysis plug-ins* which typically implement some property analysis on some mining result. For example, for Petri nets there is a plug-in which constructs place invariants, transition invariants, and a coverability graph. However, there are also analysis plug-ins to compare a log and a model (i.e., conformance checking) or a log and an LTL formula. Moreover, there are analysis plug-ins related to performance measurement (e.g., projecting waiting times onto a Petri net).
- *Conversion plug-ins* implement conversions between different data formats, e.g., from EPCs to Petri nets or from Petri nets to BPEL.
- *Log filter plug-ins* implement different ways of "massaging" the log before applying process mining techniques. For example, there are plug-ins to select different parts of the log, to abstract from infrequent behavior, clean the log by removing incomplete cases, etc.

In this paper we do not elaborate on the architecture and implementation framework for plug-ins (for this we refer to [7]). Instead we focus on the functionality provided by the many new plug-ins in ProM 4.0.

The remainder of this paper is organized as follows. Section 2 provides an overview of process mining and briefly introduces the basic concepts. Section 3 describes the "teleclaims" process of an Australian insurance company. A log of this process is used as a running example and is used to explain the different types of process mining: Discovery (Section 4), Conformance (Section 5), and Extension (Section 6). Section 7 briefly mentions additional functionality such as verification and model transformation. Section 8 concludes the paper.

2 Overview

The idea of process mining is to discover, monitor and improve *real* processes (i.e., not assumed processes) by extracting knowledge from event logs. Today many of the activities occurring in processes are either supported or monitored by information systems. Consider for example ERP, WFM, CRM, SCM, and PDM systems to support a wide variety of business processes while recording well-structured and detailed event logs. However, process mining is not limited to information systems and can also be used to monitor other operational processes or systems. For example, we have applied process mining to complex X-ray machines, high-end copiers, web services, wafer steppers, careflows in hospitals, etc. All of these applications have in common that *there is a notion of a process* and that *the occurrence of activities are recorded in so-called event logs.*

Assuming that we are able to log events, a wide range of *process mining techniques* comes into reach. The basic idea of process mining is to learn from observed executions of a process and can be used to (1) *discover* new models (e.g., constructing a Petri net that is able to reproduce the observed behavior), (2) check the *conformance* of a model by checking whether the modeled behavior matches the observed behavior, and (3) *extend* an existing model by projecting

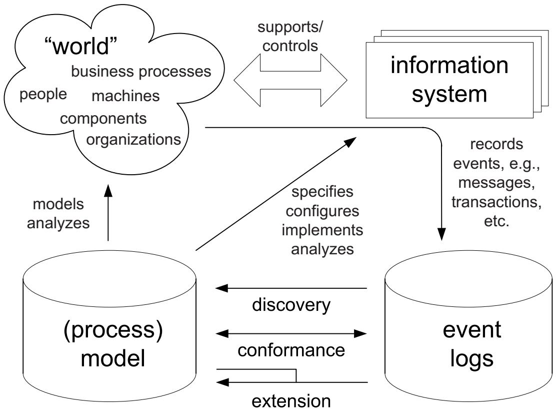


Fig. 1. Overview showing three types of process mining supported by ProM: (1) Discovery, (2) Conformance, and (3) Extension

information extracted from the logs onto some initial model (e.g., show bottlenecks in a process model by analyzing the event log). All three types of analysis have in common that they assume the existence of some *event log*. Figure 1 shows the three types of process mining. Each of these is supported by ProM through various plug-ins as will be shown in the remainder using a running example.

3 Running Example

As a working example, we consider the "teleclaims" process of an Australian insurance company described in [2]. This process deals with the handling of inbound phone calls, whereby different types of insurance claims (household, car, etc.) are lodged over the phone. The process is supported by two separate call centres operating for two different organizational entities (Brisbane and Sydney). Both centres are similar in terms of incoming call volume (approx. 9,000 per week) and average total call handling time (550 seconds), but different in the way call centre agents are deployed, underlying IT systems, etc. The teleclaims process model is shown in Figure 2. The two highlighted boxes at the top show the subprocesses in both call centres. The lower part describes the process in the back-office.

This process model is expressed in terms of an Event-Driven Process Chain (EPC) (see [8] for a discussion on the semantics of EPCs). For the purpose of the paper it is not necessary to understand the process and EPC notation in any detail. However, for a basic understanding, consider the subprocess corresponding to the call centre in Brisbane. The process starts with *event* "Phone call received". This event triggers *function* "Check if sufficient information is available". This function is executed by a "Call Center Agent". Then a choice is made. The circle represents a so-called *connector*. The "x" inside the connector and the two outgoing arcs indicate that it is an exclusive OR-split (XOR). The

XOR connector results in event “Sufficient information is available” or event “Sufficient information is not available”. In the latter case the process ends. If the information is available, the claim is registered (cf. function “Register claim” also executed by a “Call Center Agent”) resulting in event “Claim is registered”. The call centre in Sydney has a similar subprocess and the back-office process should be self-explaining after this short introduction to EPCs. Note that there are three types of split and join connectors: AND, XOR, and OR, e.g., in the back-office process there is one AND-split (∧) indicating that the last part is executed in parallel.

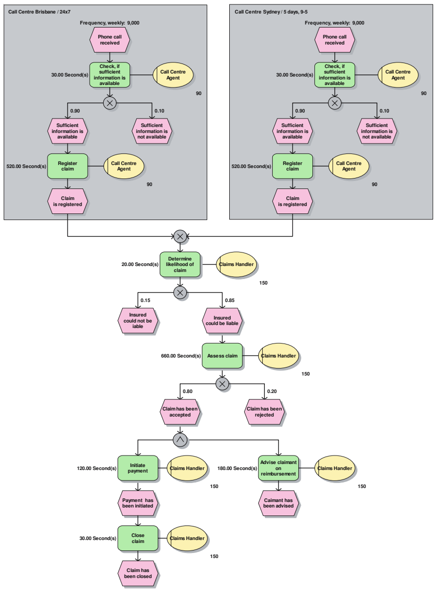


Fig. 2. Insurance claim handling EPC [2]

```
...
<ProcessInstance id="3055" description="Claim being handled">
  <AuditTrailEntry>
    <Data><Attribute name = "call centre">Sydney </Attribute>
    </Data><WorkflowModelElement>incoming claim
      </WorkflowModelElement>
    <EventType >complete</EventType>
    <Timestamp>2006-12-01T07:51:05.000+01:00</Timestamp>
    <Originator>customer</Originator>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <Data><Attribute name = "location">Sydney </Attribute>
    </Data><WorkflowModelElement>check if sufficient
      information is available</WorkflowModelElement>
    <EventType >start</EventType>
    <Timestamp>2006-12-01T07:51:05.000+01:00</Timestamp>
    <Originator>Call Centre Agent Sydney</Originator>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <Data><Attribute name = "location">Sydney </Attribute>
    </Data><WorkflowModelElement>check if sufficient
      information is available</WorkflowModelElement>
    <EventType >complete</EventType>
    <Timestamp>2006-12-01T07:51:25.000+01:00</Timestamp>
    <Originator>Call Centre Agent Sydney</Originator>
  </AuditTrailEntry>
  ...
  <AuditTrailEntry>
    <Data><Attribute name = "outcome">processed </Attribute>
    <Attribute name = "duration">1732 </Attribute>
    </Data><WorkflowModelElement>end</WorkflowModelElement>
    <EventType >complete</EventType>
    <Timestamp>2006-12-01T08:19:57.000+01:00</Timestamp>
    <Originator>Claims handler</Originator>
  </AuditTrailEntry>
</ProcessInstance>
...
```

Fig. 3. Fragment of the MXML log containing 3512 cases (process instances) and 46138 events (audit trail entries)

Figure 3 shows a fragment of the log in MXML format, the format used by ProM. In this case, the event log was obtained from a simulation using CPN Tools. Using ProMimport one can extract logs from a wide variety of systems, e.g., workflow management systems like Staffware, case handling systems like FLOWer, ERP components like PeopleSoft Financials, simulation tools like ARIS and CPN Tools, middleware systems like WebSphere, BI tools like ARIS PPM, etc., and it has also been used to develop many organization/system-specific conversions (e.g., hospitals, banks, governments, etc.). Figure 3 illustrates the typical data present in most event logs, i.e., a log is composed of process instances (i.e., cases) and within each instance there are audit trail entries (i.e., events) with various attributes. Note that it is not required that systems log all of this information, e.g., some systems do not record transactional information (e.g., just the completion of activities is recorded), related data, or timestamps. In the MXML format only the ProcessInstance (i.e., case) field and the WorkflowModelElement (i.e., activity) field are obligatory, i.e., *any event needs to be linked to a case (process instance) and an activity*. All other fields (data, timestamps, resources, etc.) are optional.

For control-flow discovery, e.g., deriving a Petri net model from an MXML file, we often focus on the ordering of activities within individual cases. In this context, a single case σ can be described by a sequence of activities, i.e., a trace $\sigma \in A^*$ where A is the set of activities. Consequently, such an abstraction of the log can be described by a multiset of traces.

4 Discovery

Process mining techniques supporting *discovery* do not assume an a-priori model, i.e., based on an event log, some model is constructed (cf. Figure 1). ProM 4.0 offers 27 mining plug-ins able to construct a wide variety of models. One of the first plug-ins was the α-miner [5] which constructs a Petri net model from an MXML log, i.e., based on an analysis of the log which does not contain any explicit process information (e.g., AND/XOR-splits/joins), a process model is derived. However, the α-miner is unable to discover complex process models. For example, it is unable to *correctly* discover the teleclaims process illustrated in Figure 2. However, ProM 4.0 has several new mining plug-ins that are able to correctly discover this process using various approaches (regions, heuristics, genetic algorithms, etc.) and representations (Petri nets, EPCs, transitions systems, heuristic nets).

Figure 4 shows a Petri net discovered by ProM. The top window shows the overall process while the second window zooms in on the first part of the discovered model. This model is behaviorally equivalent to the EPC model in Figure 2 and has been obtained using an approach which first builds a transition system (see Figure 5) and then uses extensions of the classical theory of regions [6] to construct a Petri net. ProM provides various ways to extract transition systems from logs, a plug-in to construct regions on-the-fly, and an import and export plug-in for Petrify [6] (see [3] for details).

Process mining is not limited to process models (i.e., control flow). ProM also allows for the discovery of models related to data, time, transactions, and resources. As an example, Figure 6 shows the plug-in to extract social networks from event logs using the technique presented in [1]. The social network shown in Figure 6 is constructed based on frequencies of work being transferred from one resource class to another. The diagram adequately shows that work is generated by customers and then flows via the call centre agents to the claims handlers in the back office.

It is impossible to provide an overview of all the discovery algorithms supported. However, of the 27 mining plug-ins we would like to mention the heuristics miner (Figure 7) able to discover processes in the presence of noise and

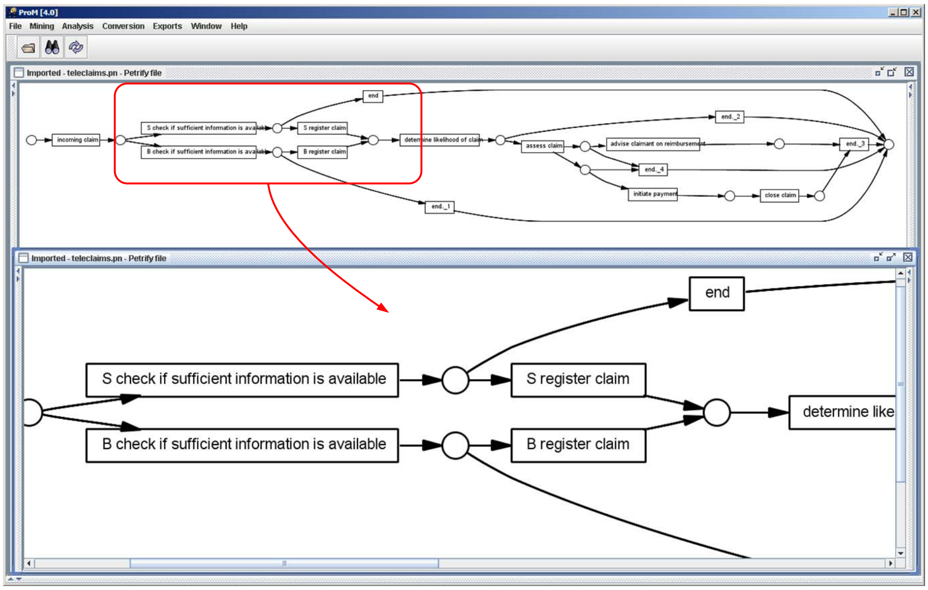
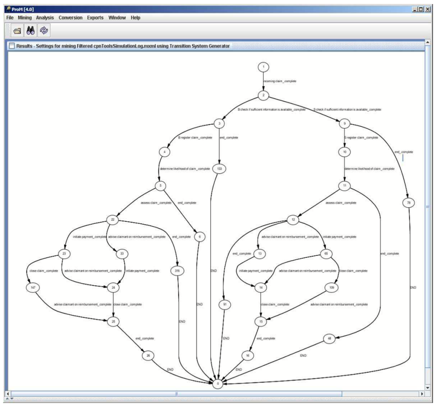
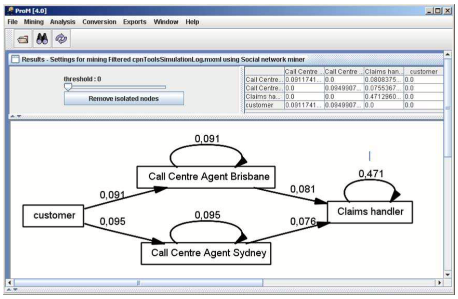

ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Imported - teleclaims.pn - Petrify file
incoming claim
S check if sufficient information is available
B check if sufficient information is available
S register claim
B register claim
end
determine likelihood of claim
end._2
assess claim
advise claimant on reimbursement
end._3
end._4
initiate payment
close claim
end._1
Imported - teleclaims.pn - Petrify file
end
S check if sufficient information is available
S register claim
determine like
B check if sufficient information is available
B register claim

Fig. 4. A Petri net discovered using ProM based on an analysis of the 3512 cases


ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Results - Settings for mining Filtered cpnToolsSimulationLog.mxml using Transition System Generator

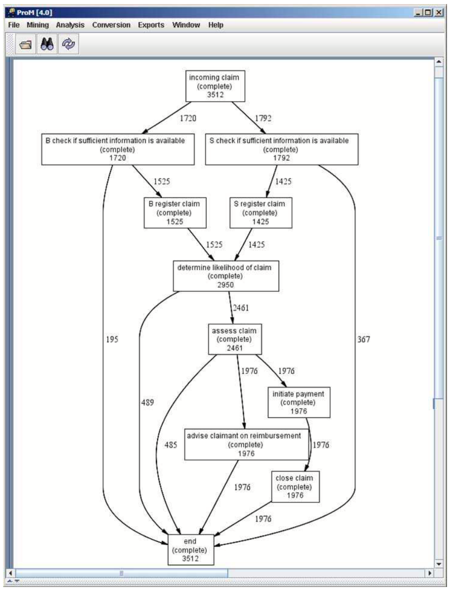

ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Results - Settings for mining Filtered cpnToolsSimulationLog.mxml using Social network miner
threshold : 0
Remove isolated nodes
0,091
Call Centre Agent Brisbane
0,471
0,091
0,081
customer
Claims handler
0,095
0,095
0,076
Call Centre Agent Sydney

	Call Centre ...	Call Centre ...	Claims han...	customer
Call Centre...	0.0911741...	0.0	0.0808375...	0.0
Call Centre...	0.0	0.0949907...	0.0755367...	0.0
Claims ha...	0.0	0.0	0.4712960...	0.0
customer	0.0911741...	0.0949907...	0.0	0.0

Fig. 5. Transition system system used to construct the Petri net in Figure 4

Fig. 6. Social network obtained using the "handover of work" metric

the multi-phase miner using an EPC representation. Both approaches are more robust than the region-based approach and the classical α-algorithm. It is also possible to convert models of one type to another. For example, Figure 8 shows the EPC representation of the Petri net in Figure 4.


ProM [4.0]
File Mining Analysis Conversion Exports Window Help
incoming claim (complete) 3512
1720
1792
B check if sufficient information is available (complete) 1720
S check if sufficient information is available (complete) 1792
1525
1425
B register claim (complete) 1525
S register claim (complete) 1425
1525
1425
determine likelihood of claim (complete) 2950
2461
assess claim (complete) 2461
195
367
489
1976
1976
initiate payment (complete) 1976
485
advise claimant on reimbursement (complete) 1976
1976
close claim (complete) 1976
1976
1976
end (complete) 3512

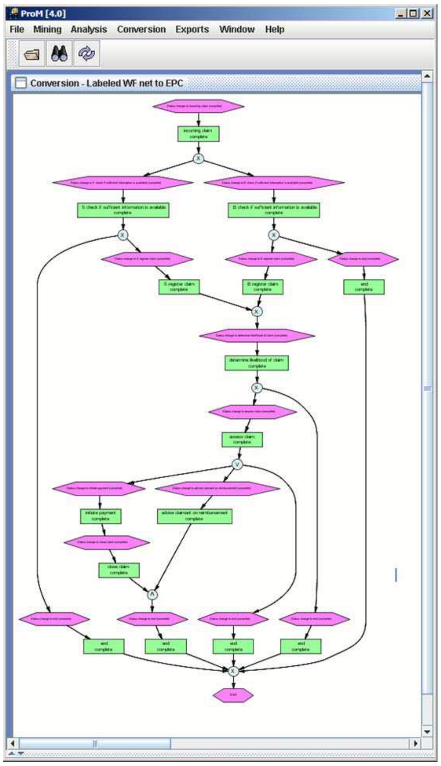
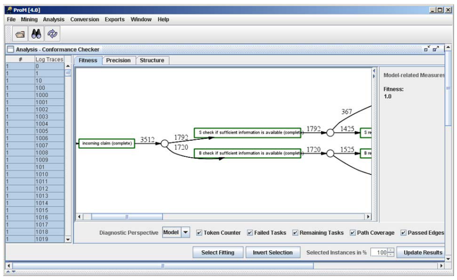

ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Conversion - Labeled WF net to EPC

Fig. 7. Heuristics net obtained by applying the heuristics miner to the log of Figure 3

Fig. 8. EPC discovered from the log in Figure 3

5 Conformance

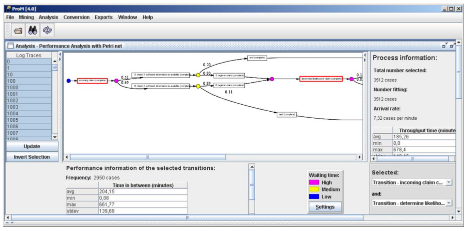
Conformance checking requires, in addition to an event log, some a-priori model. This model may be handcrafted or obtained through process discovery. Whatever its source, ProM provides various ways of checking whether reality conforms to such a model. For example, there may be a process model indicating that purchase orders of more than one million Euro require two checks. Another example is the checking of the so-called “four-eyes principle”. Conformance checking may be used to detect deviations, to locate and explain these deviations, and to measure the severity of these deviations. ProM 4.0 also supports conformance checking, i.e., comparing an a-priori model with the observed reality stored in some MXML log. For example, we could take the discovered model shown in Figure 4 and compare it with the log shown in Figure 3 using the conformance checking plug-in in ProM. Figure 9 shows the result. This analysis shows that the fitness of the model is 1.0, i.e., the model is able to “parse” all cases. The conformance checker also calculates metrics such as behavioral appropriateness (i.e., precision) and structural appropriateness [9] all indicating that the discovered model is indeed a good reflection of reality. Note that, typically, conformance checking is done not with respect to a discovered model, but with respect to some normative/descriptive hand-crafted model. For example, given an event log obtained from the real teleclaims process it would be interesting to detect potential deviations from the process model in Figure 2. In case that there is not a complete a-priori process model but just a set of requirements (e.g., business rules), ProM's LTL checker can be used.


ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Analysis - Conformance Checker
Fitness Precision Structure
incoming claim (complete)
S check if sufficient information is available (complete)
B check if sufficient information is available (complete)
3512
1792
1720
367
1425
1525
Diagnostic Perspective Model
Token Counter Failed Tasks Remaining Tasks Path Coverage Passed Edges
Select Fitting Invert Selection Update Results

#	Log Traces
1	0
1	1
1	10
1	100
1	1000
1	1001
1	1002
1	1003
1	1004
1	1005
1	1006
1	1007
1	1008
1	1009
1	101
1	1010
1	1011
1	1012
1	1013
1	1014
1	1015
1	1016
1	1017
1	1018
1	1019

Model-related Measures
Fitness:
1.0

Selected Instances in % 100



Process information:

Total number selected:
3512 cases

Number fitting:
3512 cases

Arrival rate:
7,32 cases per minute

	Throughput time (minute
avg	185,26
min	0,0
max	678,4

Performance information of the selected transitions:

Frequency: 2950 cases

	Time in between (minutes)
avg	204,15
min	0,88
max	661,77
stdev	139,69

Waiting time:

🟪	High
🟨	Medium
🟦	Low

Selected:
Transition - incoming claim c...
and:
Transition - determine likeliho...

Fig. 9. Conformance checker

Fig. 10. Performance analyzer

6 Extension

For model *extension* it is also assumed that there is an initial model (cf. Figure 1). This model is extended with a new aspect or perspective, i.e., the goal is not to check conformance but to enrich the model with performance/time aspects, organizational/resource aspects, and data/information aspects. Consider for example a Petri net (either discovered, hand-crafted, or resulting from some model transformation) describing a process which is also logged. It is possible to enrich the Petri net using information in the log. Most logs also contain information about resources, data, and time. ProM 4.0 supports for example decision mining, i.e., by analyzing the data attached to events and using classical decision tree analysis, it is possible to add decision rules to the Petri net (represented as conditions on arcs). Information about resources (Originator field in the MXML log) can be analyzed and used to add allocation rules to a Petri net. Figure 10 shows a performance analysis plug-in which projects timing information on places and transitions. It graphically shows the bottlenecks and all kinds of performance indicators, e.g., average/variance of the total flow time or the time spent between two activities. The information coming from all kinds of sources can be stitched together and exported to CPN Tools, i.e., ProM is able to turn MXML logs into colored Petri nets describing all perspectives (control-flow, data, time, resources, etc.). CPN Tools can then be used to simulate the process without adding any additional information to the generated model.

7 Additional Functionality

It is not possible to give a complete overview of all 142 plug-ins. The figures shown in previous sections reflect only the functionality of 7 plug-ins. However, it is important to note that the functionality of ProM is not limited to process mining. ProM also allows for *model conversion*. For example, a model discovered in terms of a heuristic net can be mapped onto an EPC which can be converted into a Petri net which is saved as a YAWL file that can be uploaded in the workflow system YAWL thereby directly enacting the discovered model. For some of the models, ProM also provides *analysis* plug-ins. For example, the basic Petri net analysis techniques (invariants, reachability graphs, reduction rules, S-components, soundness checks, etc.) are supported. There are also interfaces to different analysis (e.g., Petrify, Fiona, and Woflan) and visualization (e.g., FSMView and DiaGraphica) tools.

8 Conclusion

ProM 4.0 consolidates the state-of-the-art of process mining. It provides a plug-able environment for process mining offering a wide variety of plug-ins for process discovery, conformance checking, model extension, model transformation, etc. ProM is open source and can be downloaded from `www.processmining.org`. Many of its plug-ins work on Petri nets, e.g., there are several plug-ins to discover Petri nets using techniques ranging from genetic algorithms and heuristics to regions and partial orders. Moreover, Petri nets can be analyzed in various ways using the various analysis plug-ins.

Acknowledgements. The development of ProM is supported by EIT, NWO-EW, the Technology Foundation STW, and the IOP program of the Dutch Ministry of Economic Affairs.

References

1. van der Aalst, W.M.P., Reijers, H.A., Song, M.: Discovering Social Networks from Event Logs. Computer Supported Cooperative work 14(6), 549–593 (2005)
2. van der Aalst, W.M.P., Rosemann, M., Dumas, M.: Deadline-based Escalation in Process-Aware Information Systems. Decision Support Systems 43(2), 492–511 (2007)
3. van der Aalst, W.M.P., Rubin, V., van Dongen, B.F., Kindler, E., Günther, C.W.: Process Mining: A Two-Step Approach using Transition Systems and Regions. BPM Center Report BPM-06-30, (2006) `BPMcenter.org`
4. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow Mining: A Survey of Issues and Approaches. Data. and Knowledge Engineering 47(2), 237–267 (2003)
5. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data. Engineering 16(9), 1128–1142 (2004)

6. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri Nets from Finite Transition Systems. IEEE Transactions on Computers 47(8), 859–882 (1998)
7. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM framework: A New Era in Process Mining Tool Support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005)
8. Kindler, E.: On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. Data and Knowledge Engineering 56(1), 23–40 (2006)
9. Rozinat, A., van der Aalst, W.M.P.: Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In: Bussler, C., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 163–176. Springer, Heidelberg (2006)

dmcG: A Distributed Symbolic Model Checker Based on GreatSPN

Alexandre Hamez, Fabrice Kordon, Yann Thierry-Mieg, and Fabrice Legond-Aubry

Université P. & M. Curie,
LIP6 - CNRS UMR 7606
4 Place Jussieu, 75252 Paris cedex 05, France
Alexandre.Hamez@lip6.fr, Fabrice.Kordon@lip6.fr, Yann.Thierry-Mieg@lip6.fr, Fabrice.Legond-Aubry@lip6.fr

Abstract. We encountered some limits when using the GreatSPN model checker on life-size models, both in time and space complexity. Even when the exponential blow-up of state space size is adequately handled by the tool thanks to the use of a *canonization* function that allows to exploit system symmetries, time complexity becomes critical. Indeed the canonization procedure is computationally expensive, and verification time for a single property may exceed 2 days (without exhausting memory).

Using the GreatSPN model-checking core, we have built a distributed model-checker, dmcG, to benefit from the aggregated resources of a cluster. We built this distributed version using a flexible software architecture dedicated to *parallel and distributed* model-checking, thus allowing full reuse of GreatSPN source code at a low development cost. We report performances on several specifications that show we reach the theoretical linear speedup w.r.t. the number of nodes. Furthermore, through intensive use of multi-threading, performances on multi-processors architectures reach a speedup linear to the number of processors.

Keywords: GreatSPN, Symbolic Reachability Graph, Distributed Model Checking.

1 Introduction

If we want model checking to cope with industrial-size specifications, it is necessary to be able to handle large state spaces. Several techniques do help:

i compact encoding of a state space using decision diagrams; these techniques are called symbolic[1] model checking [3,1],

ii equivalence relation based representation of states that group numerous *concrete* states of a system into a *symbolic* state [2]; these techniques are also called *symbolic*,

iii executing the model checker on a distributed architecture [4,5,6,7,8].

[1] The word *symbolic* is associated with two different techniques. The first one is based on state space encoding and was introduced in [1]. The second one relies on set-based representations of states having similar structures and was introduced in [2].

These three techniques can be stacked. This was experimented for *(i)* and *(ii)* in [9].

GreatSPN [10] is a well known tool implementing technique *(ii)* in its model checking kernel thanks to the use of Symmetric Nets [2] for input specifications. We have successfully used it for many studies requiring to analyse complex systems with more than 10^{18} concrete states and an exponential gain between the concrete reachability graph and the symbolic reachability graph [11].

However, these studies and another one dedicated to intelligent transport systems [12] revealed two problems. First, generation of the symbolic reachability graph requires a canonical representation of states to detect already existing ones. This is a known problem that requires CPU time. So, even if memory consumption is reduced, CPU load becomes a problem.

The second problem deals with an implementation constant that prevented us to handle more than 12 million symbolic states in the version we had (in the first study we mentioned, this was the equivalent to 10^{18} concrete states).

So, to handle larger systems, the idea is to use the aggregated resources of a cluster and thus, merge techniques *(ii)* and *(iii)* by implementing a distributed model checker for Symmetric Nets able to generate faster larger state spaces.

This paper presents how we built dmcG, a distributed model checker based on the GreatSPN model-checking core. We did not change the GreatSPN sources files. They were encapsulated in a higher-level program called libdmc [13]. This library is dedicated to parallelization and distribution of model checkers and orchestrates services provided by GreatSPN to enable a distributed execution.

The paper is organized as follow. After a survey of related works in Section 2, Section 3 explains how we built dmcG atop libdmc using GreatSPN. Then, Section 4 presents performances of dmcG and discusses about them.

2 Related Work

Several attempts at proposing a distributed model-checker have been made. In [6] the authors implemented a distributed version of Spin. The problem however, was that the main state space reduction technique of Spin, called partial order reduction, had to be re-implemented in a manner that degrades its effectiveness as the number of hosts collaborating increases. Thus performances, reported up to 4 hosts in the original paper, were reported to actually not scale well on a cluster (see Nasa's case study in [14]). Another effort to implement a distributed Spin is DivSpin [4]. However, they chose to re-implement a Promela engine rather than using Spin's source code. As a result their sequential version is at least twice as slow as sequential Spin in it's most degraded setting with optimizations deactivated. And any further improvements of the Spin tool will not profit their implementation.

An effort that has met better success is reported in [5] for a distributed version of the Murphi verifier from Stanford. Murphi exhibits a costly canonization procedure. The original implementation in [5] was built on top of specific Berkeley NOW hardware[3], which limits its portability. A more recent implementation [7] is based on MPI [15],

[2] Formerly known as Well-Formed Nets [2], a class of High-level Petri Nets.
[3] The Berkeley Network of Workstations.

however it is limited to two threads per hosts, one handling the network and one for computation of the next state function. Our work is however comparable to that effort in terms of design goals: reuse of existing code over a network of multi-processor machines, a popular architecture due to its good performance/cost ratio. The good results reported by these Murphi-based tools with slightly sublinear speedup over the number of hosts encourage further experimentation in this direction.

We can also cite [8] which is an effort to distribute the generation of (but theoretically not limited to) LOTOS specifications. They report near linear speedups.

3 Building dmcG

Our strategy is to build a model checker based on the GreatSPN core, thus reusing its implementation of the symbolic reachability graph. So, we let GreatSPN compute successors but its execution is handled by the libdmc library. This one distributes the state space over the nodes of a cluster and manages all the networking and multi-threaded aspects of the state space generation.

To distribute *memory load*, we assign an *owner node* to each state, responsible for storing it. We use a static localization function that, for each state, designates a host. Note that this function should have a homogeneous distribution to ensure *load balancing*: we chose MD5 as it is known to provide a good distribution for any kind of input.

To distribute *CPU load*, each node has an active component that computes successors. This is the computationally most expensive task in the model-checking procedure, particularly because of the canonization algorithms of GreatSPN.

3.1 Architecture of dmcG

dmcG is designed using a component based approach. Each node instantiates the components shown on figure 1:

1. The *StateManager* is a passive storage component instanciated once per node, that stores the states owned by the node. The behavior of *processState* is to determine

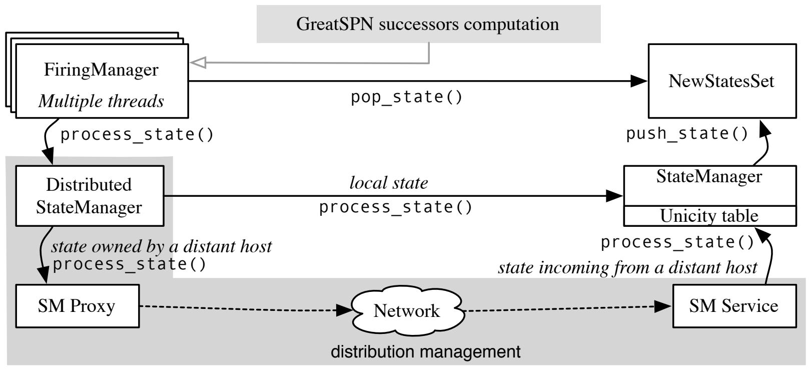


Fig. 1. dmcG architecture

whether a state is new or not. If a state is new, it is stored in an unicity table and also pushed into the *NewStatesSet*, otherwise, it is discarded.

2. The *NewStatesSet* is a protected passive state container, used to store states that have not been fully explored. Each node has a single occurrence of this component, shared by the active computing threads.
3. The *FiringManager* represents an active thread used to compute successors by calling GreatSPN successors functions. Several instances of this component are instanciated on each node to allow full use of multi-processor machines, and to overlap network latency. As shown in figure 1, each *FiringManager* instance repeatedly pops a state from the *NewStatesSet*, computes its successors and passes them to the *DistributedStateManager*. One privileged *FiringManager* initiates the computation by processing the initial state(s) on a master node.
4. The *DistributedStateManager* is responsible for forwarding the states to their owner, using the localization function. This component deals with a set of *StateManager Proxies* representing distant *StateManager* (*proxy* design pattern [16]). When a state is not owned by the node which computed it, it is transmitted to its owner through its proxy/service pair. Each node instantiates one *DistributedStateManager* component for localization purposes, one proxy per distant node, and one service to receive incoming states computed on other nodes.

3.2 Interaction with GreatSPN

The first step was to identify the core functions of GreatSPN which compute successors of a state. This work was made easier thanks to a previous similar effort aimed at integrating LTL model-checking capacity in GreatSPN, using the Spot library [17]. The interface we defined corresponds to a labeled automata: we extracted functions to obtain the initial state, and a successor function returning the set of successors of a given state (using an iterator). An additional labeling function was extracted to allow labeling of states satisfying a given criterion (deadlock state, state verifying some arbitrary state property...).

This simple interface is defined to minimize dependencies from libdmc to a given formalism. We did not redevelop any algorithms for successor generation in libdmc: this is essential to allow the reuse of existing efficient implementations of state space generators, and their usually quite complex data representation. The (existing) algorithms related to state representation algorithms are cleanly separated from the distribution related algorithms. The use of a canonization function by GreatSPN is thus transparent for libdmc. It manipulates compact state representations, in which each "symbolic state" actually represents an equivalence class of states of the concrete reachability graph.

The states are handled in an opaque manner by libdmc : they are seen as a simple block of contiguous memory, at interaction points between model-checking engine and libdmc, thus reducing dependencies between the model-checker and libdmc. This choice of retaining a raw binary encoding of states allows a low overhead of the library, but forces to operate over a homogeneous hardware configuration so that all nodes have a common interpretation of the state data. In any case, a layer integrating machine independent state encodings (i.e. XDR) could be added, but has not been implemented.

libdmc is intensely multi-threaded to allow the better use of modern hardware architectures, thus posing some problems when integrating legacy C code such as GreatSPN. GreatSPN is inherently non-reentrant, due to numerous global variables. The solution adopted consists in compiling the tool as a shared library that can be loaded and dynamically linked into. Simply copying the resulting shared object file in different file locations allows to load it several times into different memory spaces. This is necessary as threads usually share memory space. Each successor computation thread is assigned a separate copy of the GreatSPN binary.

We should emphasize the fact that a complete rewriting of GreatSPN was not possible since it is made of complex algorithms stacked in an architecture that has more than ten years of existence. Furthermore It wouldn't have validated the fact that the libdmc can interact with legacy code.

3.3 Verification of Safety Properties

The verification of safety or reachability properties is an important task since many aspects of modeled systems can be checked by such properties, and it is a necessary basis to handle more complex temporal logics. Finding reachable states that verify or not a property is an easy task once the state space is generated. Each node simply examines the states it has stored. This provides a yes/no answer to reachability queries. However, to let users determine how errors happened in their specifications, we need to provide a witness trace (or 'counter-example') leading to the target state. We provide a minimal counter-example in order to simplify the debuging task for the designer.

During the construction of the reachable state set, arcs are not stored however. And unfortunately, no predecessor function is available in the GreatSPN core. The approach used is to store during the construction in each state its distance to the initial state along the shortest path possible. Due to nondeterminism, the first path found to a given state is not necessarily the shortest, thus the distance may be updated if a state is reached later by a shorter path. In such a case, we have to recompute successors of the state to update their own distance, etc... Since this scheme can be costly, to avoid its occurrence as much as possible, states are popped from each node's *NewStatesSet* in ascending distance. This scheme does not introduce additional synchronization among nodes, and helps to maintain an overall approximative BFS state exploration.

After the generation of the whole state space, we build an index on each node that orders states by depth. It's not a CPU intensive task since we only need to iterate once on the state space (it only takes a few seconds). The drawback is that the size of this index increases with state space depth.

Once this index is built a master node controls the construction of the counter-example which leads to a state s in error at a distance n. The master asks to each slave a predecessor of s at a distance $n-1$. Slaves compute this predecessor by iterating over all states at depth $n-1$, using the index. The first state whose successor is s is sent to the master and the iteration is stopped. Then the master stores the first received predecessor in the counter-example. This operation is performed until the initial state has been found as predecessor. We then have a minimal counter-example.

4 Experimentations

Performances results have been measured on a cluster of 22 dual Xeons hyper-threaded at 2.8GHz, with 2GB of RAM and interconnected with Gigabit ethernet. We focused our evaluations on two parameters: states distribution and obtained speedups.

The following parametric specifications have been selected:

- the Dining Philosophers, a well known academic example; it is parameterized with the number of philosophers.
- a Client-Server specification with asynchronous send of messages and acknowledgments; it is parameterized with the number of clients and servers.
- the model of the PolyORB middleware kernel [11]; it is parameterized with the number of threads in the middleware. Let us note that the analysis of this model could not be achieved for more than 17 threads in sequential generation (it took more than 40 hours).

4.1 State Distribution

A homogeneous distribution of the states over the hosts in the cluster is an important issue since it is related to load balancing of dmcG. The goal is to avoid the situation were some hosts are overloaded while others are idle. This is required to reach a linear speedup.

Therefore, we measured the number of states owned by each host, in order to validate the choice of MD5 as a localization function. We then compared these results to the theoretical mean value and noted the variation. These measures are summarized in Table 1. In this table, column 1 represents the parameter that scales up the model, column 2 the number of involved hosts, column 3 the total number of symbolic states, column 4 the theoretical mean value, column 5 the standard deviation and column 6 the standard deviation expressed in percentage of the mean value.

We obtain standard deviations that are from less than 0.1% to less than 5% of the mean values. Such results are obtained for every model we analyzed in any configuration (model parameters and number of nodes). So our expectations are met since the state space is evenly distributed over the whole cluster. However, we observe that, as a possible side effect of the MD5 checksum, it is preferable to have a number of nodes that is a power of 2, in which case the standard deviation is smaller than 1% of the mean value. Additional experiments showed that the usage of a more complex checksum like SHA-1 doesn't seem to improve or to decrease the quality of the states distribution.

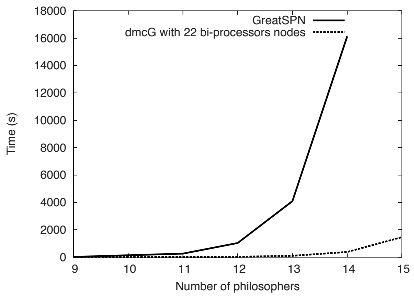
4.2 Speedups

Figures 2, 3 and 4 show the compared time of sequential and distributed generation needed for the three specifications we analyzed. They also show the speedups we obtain for the distributed generation of these models. Speedups are compared to the runtime of the standard GreatSPN tool running the same example.

dmcG has a low overhead: execution for the mono-threaded, single host version of dmcG takes within 95% to 105% of the time needed by the standard GreatSPN version

Table 1. States distribution for the Philosophers, Client-Server and PolyORB specifications

Parameter	Hosts	Symb. States	Mean	Std. deviation	Percentage
Dining Philosophers					
12 philosophers	4	347 337	86 834	427	0.5%
15 philosophers	4	12 545 925	3 136 481	792	< 0.1%
12 philosophers	22	347 337	15 788	689	4.4%
15 philosophers	22	12 545 925	570 269	24 179	4.2%
Client-Server					
100 processes	4	176 851	44 213	202	0.5%
400 processes	4	10 827 401	2 706 850	1327	< 0.1%
100 processes	20	176 851	8 843	316	3.57%
400 processes	20	10 827 401	541 370	17 438	3.22%
PolyORB middleware					
11 threads	16	3 366 471	210 404	402	0.2%
17 threads	16	12 055 899	753 494	1131	0.2%
11 threads	20	3 366 471	168 324	5393	3.2%
17 threads	20	12 055 899	602 795	19 557	3.2%
25 threads	20	37 623 267	1 881 163	60 540	3.7%



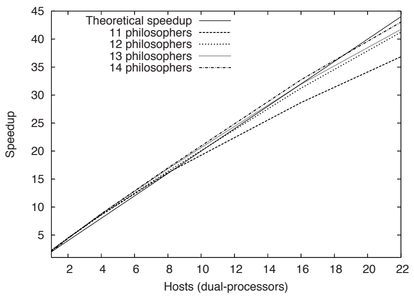
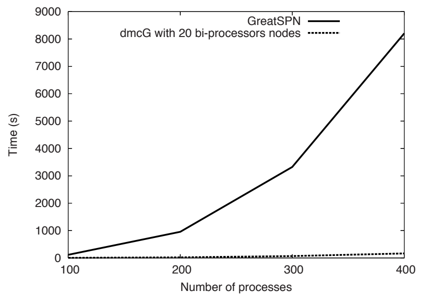


Fig. 2. Generation time (left) and speedups (right) for the Dining Philosophers specification

(these variations are due to implementation details concerning the storage of states). The local but multi-threaded version is truly twice as fast on a dual-processor machine.

The main observation is that, in many cases, the observed speedup is over the theoretical one based on the number of *processors* (two per host): we have a supra-linear acceleration factor, up to 50 with 20 bi-processors nodes (fig. 3). We observed this in near all our experiments on several models with various parameters. We attribute this to hyper-threading since the supra-linear acceleration factor was not observed on classic dual processors (without hyper-threading). This hypothesis is confirmed by [18].

The stall observed in figure 3 for the Client-Server specification parameterized with 100 processes is simply due to the fact that the execution time is very small (<5s): the specification becomes too simple to compute for a number of nodes superior to 16.



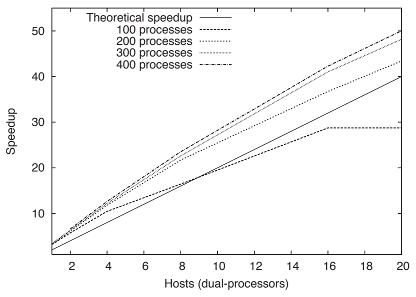
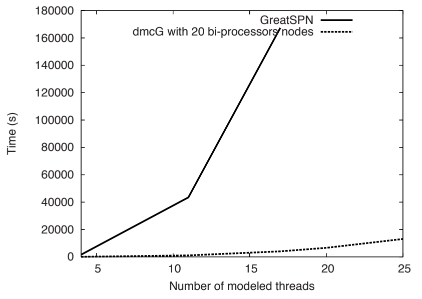


Fig. 3. Generation time (left) and speedups (right) for the Client-Server specification



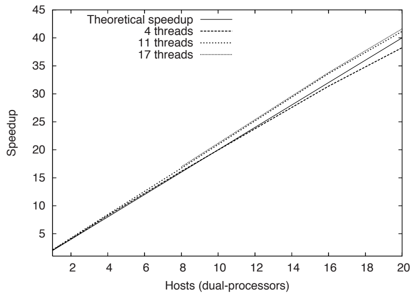


Fig. 4. Generation time (left) and speedups (right) for the PolyORB specification

4.3 Other Considerations

Another measure of interest is the network bandwidth consumption: measured bandwidths per node are from 230 KB/s for the PolyORB specification to 1.5 MB/s for the Client-Server specification, in a configuration with 20 nodes. That means that we do not use more than 30 MB/s of total bandwidth in this configuration, which any modern switch should handle easily.

We also observe that the larger the state space, the more efficient dmcG is. We impute this to the fact that there are more chances for a state to have at least a successor that is then distributed to another host, leading the *NewStatesSet* of each node to never be empty during the computation (which would make idle hosts).

Finally, a preliminary campaign on a larger cluster with 128 dual-processors nodes shows that dmcG scales up very well: we continue to observe a growing linear speedup with large models, as well as a homogeneous distribution.

5 Conclusion

In this paper, we presented dmcG, a distributed model checker working on a symbolic state space. The goal is to stack two accelerating techniques for model checking in order to get a more powerful tool. As a basis for the core functions of this distributed model checker, we used GreatSPN implementation without changing it. It was connected to a library dedicated to the distribution of model checking: libdmc.

Performances on several models, including industrial-like case study (the verification of a middleware's core) are almost optimal. We observe a nearly linear speedup with, in some favorable cases, a supra-linear speedup (due to the intensive use of both distribution and multi-threading). Using dmcG, we can push memory and CPU capacity of our model-checker one to two orders of magnitude further.

So far, our model checker basically manages safety properties and, when a property is not verified, provides an execution path to the faulty state. It takes as input AMI-nets or native GreatSPN format models, with additional parameters to specify properties. The multi-node version still requires some skill to install and configure, but we plan to make it available soon. The multi-threaded single node version is immediately useful to owners of a bi-processor machine.

We are currently challenging libdmc to handle the verification of temporal logic formulae.

References

1. Burch, J., Clarke, E., McMillan, K.: Symbolic model checking: 10^{20} states and beyond. Information and Computation (Special issue for best papers from LICS90) 98(2), 153–181 (1992)
2. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In: Jensen, K., Rozenberg, G. (eds.) Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN'90), Reprinted in High-Level Petri Nets, Theory and Application, Springer, Heidelberg (1991)
3. Ciardo, G., Luettgen, G., Siminiceanu, R.: Efficient symbolic state-space construction for asynchronous systems. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 103–122. Springer, Heidelberg (2000)
4. Barnat, J., Forejt, V., Leucker, M., Weber, M.: DivSPIN - a SPIN compatible distributed model checker. In: Leucker, M., van de Pol, J., eds.: 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05), Lisbon, Portuga (2005)
5. Stern, U., Dill, D.L.: Parallelizing the Murφ verifier. In: Proceedings of the 9th International Conference on Computer Aided Verification, pp. 256–278. Springer, Heidelberg (1997)
6. Lerda, F., Sisto, R.: Distributed-memory model checking with SPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) Theoretical and Practical Aspects of SPIN Model Checking. LNCS, vol. 1680, Springer, Heidelberg (1999)
7. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. In: Valmari, A. (ed.) Model Checking Software. LNCS, vol. 3925, pp. 108–125. Springer, Heidelberg (2006)
8. Garavel, H., Mateescu, R., Smarandache, I.: Parallel State Space Construction for Model-Checking. vol. 2057 (2001)
9. Thierry-Mieg, Y., llié, J.M., Poitrenaud, D.: A symbolic symbolic state space representation. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 276–291. Springer, Heidelberg (2004)
10. GreatSPN V2.0 (2007) http://www.di.unito.it/~greatspn/index.html
11. Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Baarir, S., Vergnaud, T.: On the Formal Verification of Middleware Behavioral Properties. In: 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04), pp. 139–157. Elsevier, Amsterdam (2004)

12. Bonnefoi, F., Hillah, L., Kordon, F., Frémont, G.: An approach to model variations of a scenario: Application to Intelligent Transport Systems. In: Workshop on Modelling of Objects, Components, and Agents (MOCA'06), Turku, Finland (2006)
13. Hamez, A., Kordon, F., Thierry-Mieg, Y.: libDMC: a library to Operate Efficient Distributed Model Checking. In: Workshop on Performance Optimization for High-Level Languages and Libraries - associated to IPDPS'2007, Long Beach, California, USA, IEEE Computer Society, Washington (2007)
14. Rangarajan, M., Dajani-Brown, S., Schloegel, K., Cofer, D.D.: Analysis of distributed spin applied to industrial-scale models. In: Graf, S., Mounier, L. (eds.) Model Checking Software. LNCS, vol. 2989, pp. 267–285. Springer, Heidelberg (2004)
15. Message Passing Interface (2007) http://www.mpi-forum.org/
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc.,, Boston, MA, USA (1995)
17. Duret-Lutz, A., Poitrenaud, D.: Spot: an extensible model checking library using transition-based generalized Büchi automata. In: Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04), Volendam, The Netherlands, pp. 76–83. IEEE Computer Society Press, Washington (2004)
18. Koufaty, D., Marr, D.T.: Hyperthreading Technology in the Netburst Microarchitecture. IEEE Micro 23(2), 56–65 (2003)

Workcraft: A Static Data Flow Structure Editing, Visualisation and Analysis Tool

Ivan Poliakov, Danil Sokolov, and Andrey Mokhov

Microelectronics System Design Research Group, School of EECE,
University of Newcastle, Newcastle upon Tyne, UK
{ivan.poliakov,danil.sokolov,andrey.mokhov}@ncl.ac.uk

Abstract. Reliable high-level modeling constructs are crucial to the design of efficient asynchronous circuits. Concepts such as static data flow structures (SDFS) considerably facilitate the design process by separating the circuit structure and functionality from the lower-level implementation details.

Aside from providing a more abstract, higher level view, SDFS allow for efficient circuit analysis that is done by converting it to a Petri Net preserving behavioural equivalence. Once the equivalent Petri Net is obtained, existing theoretical and tool base can be applied to perform the model verification.

However, recent advances in SDFS design were largely theoretical. There are no practical software tools available which would allow working with different SDFS models in a consistent way and provide means for their analysis and comparison.

This paper presents a tool which aims to provide a common, cross-platform environment to assist with aforementioned tasks. The tool offers a GUI-based framework for visual editing, real-time simulation, animation and extendable analysis features for different SDFS types. The models themselves, as well as the supporting tools, are implemented as plug-ins.

1 Introduction

For a long time, token-flow based models, such as Petri nets, have been used as a major tool for modelling and analysis of concurrent systems. Many higher level models were developed for numerous applications; one of them is a static data flow structure which is used for data path modelling in asynchronous circuit design.

Static dataflow structure (SDFS) [12] is a directed graph with two types of nodes which model registers and combinational logic. The nodes are connected by edges which represent data channels. Registers can contain tokens, which represent the data values. The tokens can propagate in the direction of graph edges, thus modelling the propagation of data in the underlying data path.

First attempts to model asynchronous data paths using SDFS were based on mostly intuitive and somewhat naïve analogy to RTL (register transfer level) models used in synchronous design. Having been designed this way, they failed to take into account some of asynchronous circuit functionality specifics, leading to considerable limitations. There are several notable recent developments in this field, extending the basic SDFS semantics proposed in [13], such as spread token [12], anti-token [8], and counterflow [6] models, each of them allowing to model a significantly wider class of circuits.

The second chapter of this paper explains reasons which lead to the necessity of development of the presented tool. The third chapter summarizes the tool's architecture: software components, their function and interaction. The fourth chapter presents an overview of the software technologies which were used to develop the tool, with explanation of why they were chosen. The fifth chapter demonstrates the functionality of the user interface, and the sixth chapter presents an example of how Workcraft can be used to perform an analysis of a high-level model.

2 Motivation

It proves to be very hard [12] to analytically determine which model would be most appropriate for a particular class of circuits, or to compare its properties with other models; it is much easier to discover practical advantages or disadvantages if the actual modelling can be done.

This generated a need for a software tool which would provide means to integrate several different models into one consistent framework, thus enabling a circuit designer to effectively test and compare them. The tool was designed from the start to provide two key features: easy extensibility and robust cross-platform operation. To deal with evolving nature of the available models, the tool is based on *plug-in* architecture, so that introducing modifications to the existing models, or introduction of a new model is done independently of the framework. To enable efficient and intuitive document editing, a flexible vector graphics based visualisation system was designed.

This tool is the first to address the issue of supporting token-based models for asynchronous datapath modelling and design. The complex nature of concurrency in such designs, including aspects of early evaluation and Or-causality [16], presses the need for such a tool.

3 Architecture

The tool is composed of a base *framework* and three principal plug-in types as shown in Figure 1. The thick arrows on the figure depict the fact that the models, tools, and components are implemented externally and are loaded into the framework at run-time. The thinner arrows represent interaction between framework modules, i.e. the fact that one particular module is able to use the functionality provided by another one. For example, the tools can access the scripting engine, and the tools themselves can be used by scripts; the editor uses functions provided by the visualisation module, however the module itself is not aware of editor's existence.

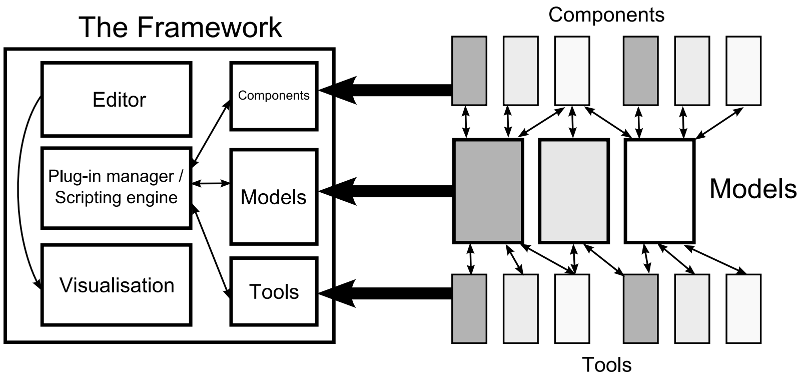


Fig. 1. Framework-plugin interaction

The framework consists of three functionally separate modules: core plug-in management and scripting system, which is referred to as the *server*, the GUI-based *editor*, and the vector graphics *visualisation* system.

The server module is the fundamental part of the framework. It performs automatic plug-in management (loading external plug-in classes, and grouping them using the reported model identifiers). It also provides a scripting engine, based on Jython [4], which is used to provide further run-time customization flexibility.

The editor provides visual environment for efficient model design. Its functions include rendering a document view using the visualisation system; viewport scaling and panning; moving and connecting components; group move, delete and copy/paste operations. The editor also supports auxiliary features, such as "snap-to-grid" function (which restricts component coordinates to intersection points of grid lines), thus enabling the user to get desired alignment of the components with ease.

The vector graphics visualisation system is designed to provide two types of output: interactive visualisation, which is implemented using OpenGL hardware acceleration, and graphics export, which renders the document as an SVG [3] file. Both visualisation methods support a common set of drawing functions, which includes drawing of lines, polylines, Bezier-curves, text and arbitrary shapes with customizable fill and outline styles. The two methods produce nearly identical result, so the developer only has to implement his or her component's drawing routine once for both interactive and external visualisation. The exported graphics provide measurements in real-world units, so that the effort of using them in printed material is minimal. The editor grid cells also have explicitly defined real-world size, thus document's final look on paper is always well-defined.

The concept of *model* is central to the software's plug-in architecture; the other plug-in types identify themselves as belonging to one or several models. The model plug-in manages simulation behaviour, handles component creation, deletion and connection operations, as well as performing document validation. Each model type defines a model identifier in the form of UUID (universally unique identifier), which is used by other plug-in types to report supported models.

Component plug-in type defines individual nodes, such as registers and combinational logic. Components define the model-related data elements, visualisation features, user-editable properties, and serialization mechanism.

Tool plug-ins define operations on the model. Their functionality is not limited in any way, so this plug-in class includes everything from external file format exporters to interfaces with stand-alone model-checking tools.

4 Software Technologies

The framework itself is implemented in Java programming language. This provides reliable cross-platform functionality, as well as very easy extensibility mechanism due to language's reflection features [14]. The plug-ins are implemented as Java classes, and the only action that is needed to make a plug-in accessible from the framework is adding a single line with the class name to the plug-in manifest file. There are also some disadvantages of using Java, with the most noticeable one being performance penalty. However, there are several options to alleviate Java's performance hit on computationally complex algorithms that may be required by analysis tools. Java Native Interface can be used to create plug-in classes which can access native code; while efficient performance-wise, this approach is somewhat cumbersome, because one has to take care of maintaining dynamically loaded code objects for each platform (which are stored in the form of DLL files in Windows, SO files in Linux and so on). Moreover, the JNI API is not very easy to use. Another approach is to implement the tools as external programs, and use the Java plug-in simply as an interface to call them. This approach can also be applied to interface already existing tools.

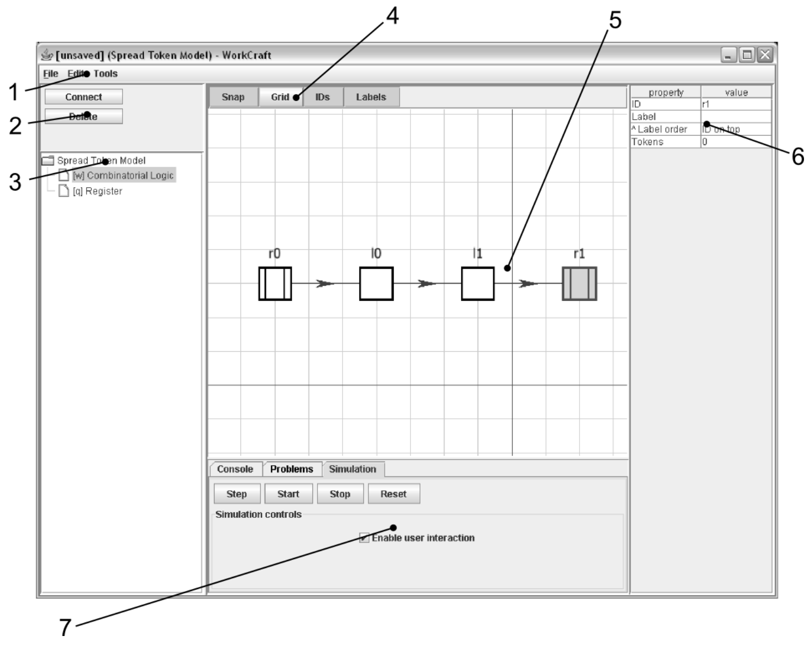
The framework supports scripting via Jython [4], a Java implementation of the Python programming language [5]. The framework automatically registers document components in Python namespace using their unique IDs, so that they can be referenced to through scripting from any place of the program.

The hardware-accelerated real-time visualisation is implemented using JOGL (Java bindings for OpenGL) [2], which gives considerable improvement in drawing performance over any software-based rendering solution, especially considering Java platform's inherent unsuitability for computationally expensive tasks. It only uses basic OpenGL functionality, and is thus compatible even with very old graphics hardware.

The XML [1] format is used to store document data. The format is very simple, flexible and modular, and the document data structures are easily mapped into it. Java platform also provides built-in support for building and parsing of XML documents. Scalable Vector Graphics [3] is used as the external graphics format because of its easy-to-use and efficient structure. For a more detailed explanation of the reasons why these choices were made, please refer to [11].

5 User Interface

The framework's main window is shown on figure 2. The *main menu* (1), beside standard file and editing operations, includes automatically selected set of tools which support current model. This selection occurs when new document is created, or a document is open from a file. The *editor commands* (2) are duplicated both in the main menu and by hotkeys. The *component list* (3) presents set of all components supported by current model, which are chosen in similar fashion to the tool set. A component can be added to the document either by dragging it from the component list, or by using a hotkey, which is optionally specified by the component. All components are further assigned numeric hotkeys from '1' to '9', corresponding to their order in component list. The *editor options bar* (4) contains toggle buttons to enable or disable certain auxiliary functions, such as display of labels, component IDs, editor grid and snap-to-grid editing mode. The *document view pane* (5) presents the document visualisation. It supports scaling (using the mouse wheel) and panning (holding right mouse button and dragging) to change the current viewport. It also supports moving components which is done by dragging them and group selection done by holding the left mouse button and dragging the selection box over desired components. The selected components can then be moved together by dragging any of them, or deleted. The *property editor* (6) displays properties of currently selected component and



property	value
ID	r1
Label	
^ Label order	ID on top
Tokens	0

Fig. 2. Main GUI window

allows editing them, such as, for example, changing the number of tokens in a Petri Net place. The *utility area* (7) holds three tabs: the console, which is used to display various information during normal execution of the program and also allows to execute script commands; the problems list which displays a list of errors which occur during execution; and the simulation control panel which allows to start, stop and reset model simulation, as well as presenting additional, model-defined simulation controls.

6 High-Level Model Analysis Within Workcraft Framework

High-level modelling introduces new challenges for the designer such as model verification and checking for certain properties e.g. deadlock-freedom. Developing a special verification theory for each high-level model is a very sophisticated problem. A possible workaround here is to transform the initial high-level model to behaviourally equivalent low-level model and thus gain access to a wide variety of theoretical and practical verification tools available for the latter. Petri nets with read-arcs [10] would be a good choice for such a low-level model as they are known for many long years and have been comprehensively studied. The chapter presents a method for conversion of an SDFS model with spread token semantics into behaviourally equivalent Petri net.

6.1 SDFS with Spread Token Semantics

An SDFS is a directed graph $G = \langle V, E, D, M_0 \rangle$, where V is a set of *vertices* (or *nodes*), $E \subseteq V \times V$ is a set of *edges* denoting the flow relation, D is a semantic domain of *data values* and M_0 is an *initial marking* of the graph. There is an edge between nodes $x \in V$ and $y \in V$ iff $(x, y) \in E$. There are two types of vertices with different semantics: *registers* R and *combinational logic nodes* L, $R \cup L = V$. The registers can contain *tokens*, thus defining the marking M of an SDFS. The tokens can be associated with data values from the semantic domain D.

The *preset* of a vertex $x \in V$ is defined as $\bullet x = \{y \in V \mid (y, x) \in E\}$ and the *postset* as $x \bullet = \{y \in V \mid (x, y) \in E\}$. Note that only registers can have empty presets and postsets. A register with empty preset is called a *source* (can be used to model system inputs), and with empty postset is called a *sink* (models system outputs).

A sequence of vertices $(z_0, z_1, ..., z_n)$ such that$(z_{i-1}, z_i) \in E$, $i = 1...n$ is called a *path* from $z_0 \in V$ (called a *start vertex*) to $z_n \in V$ (called an *end vertex*) and is denoted as $\sigma(z_0, z_n)$.

The *R-preset* of a vertex $x \in V$ is defined as $\star x = \{r \in R \mid \exists \sigma(r, x) : \sigma(r, x) \downarrow R = \{r, x\} \cap R\}$ and the *R-postset* is defined as $x \star = \{r \in R \mid \exists \sigma(x, r) : \sigma(x, r) \downarrow R = \{x, r\} \cap R\}$.

The spread token model extends the intuitive token game presented in [13]. In this model the marking is defined as a mapping $M : R \to \{0, 1\}$, i.e. a register can contain maximum one token. The *evaluation state* of an SDFS is a mapping $\Xi : L \to \{0, 1\}$ which defines if a combinational logic node $l \in L$ has computed its output ($\Xi(l) = 1$) or has not computed it yet ($\Xi(l) = 0$). A node $l \in L$ is said to be *evaluated* if $\Xi(l) = 1$ and *reset* if $\Xi(l) = 0$. Initially all combinational logic nodes are reset.

A reset combinational logic node $l \in L$ may evaluate iff $\forall k \in \bullet l \cap L$ is evaluated and $\forall q \in \bullet l \cap R$ is marked. Similarly, an evaluated combinational logic node $l \in L$ may reset iff $\forall k \in \bullet l \cap L$ is reset and $\forall q \in \bullet l \cap R$ is not marked.

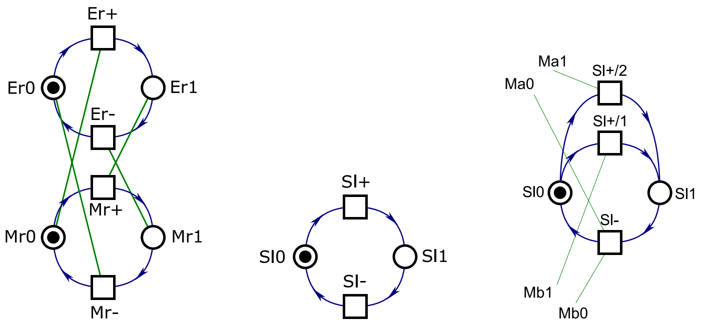
Initially all unmarked registers are *disabled* and all marked registers are *enabled*. An unmarked register $r \in R$ becomes enabled iff $\forall l \in \bullet r \cap L$ is evaluated and $\forall q \in \bullet r \cap R$ is marked. A marked register $r \in R$ becomes disabled iff $\forall l \in \bullet r \cap L$ is reset and $\forall q \in \bullet r \cap R$ is unmarked.

An enabled register $r \in R$ can be marked with a token iff $\forall q \in \star r$ is marked and $\forall q \in r\star$ is not marked. A token can be removed from a disabled register $r \in R$ iff $\forall q \in \star r$ is not marked and $\forall q \in r\star$ is marked.

6.2 Conversion of SDFS Models into Petri Nets

To convert an SDFS with spread token semantics into the corresponding behaviourally equivalent Petri net it is possible to use the following mapping algorithm. Each register $r \in R$ is mapped into a pair of elementary cycles denoting the enabling Er and marking Mr of the register. Each combinational logic node $l \in L$ is mapped into an elementary cycle denoting its evaluation state Sl. The rising and falling transitions of the elementary cycles read (by means of read-arcs) the state of other cycles according to the token game rules of the spread token model. Figures 3(a,b) shows the Petri net mappings of a register and a combinational logic node. One can see in Figure 3(a) that the two elementary cycles of a register mapping are connected by four read-arcs. It captures the fact that a register follows such a cyclic behaviour during its lifetime: *enabled & unmarked* $\to$ *enabled & marked* $\to$ *disabled & marked* $\to$ *disabled & unmarked* $\to$ *enabled & unmarked* $\to$ ... and so on.

An SDFS model with spread token semantics can be refined to take into account the *early propagation effect*. In some cases combinational logic node can evaluate its function without waiting for all of its preset nodes to become marked/evaluated. The mapping of such an early propagative combinational logic node is different and may contain several parallel transitions in the elementary cycle as shown in Figure 3(c). The figure shows the mapping of a node **l** that can evaluate after at least one of its preset registers (**a** and **b**) has been marked (parallel transitions **Sl+/1** and **Sl+/2**). And it can reset after both of the registers have been unmarked (transition **Sl-**). In general each combinational logic node can have a *propagation function* [12] associated with it. The function determines when a reset node can evaluate and when an evaluated node can reset. The propagation function is parsed into disjunctive normal form (DNF) expressions. Then each of the clauses in the obtained DNF is mapped into transition in the elementary cycle that corresponds to the node that is being processed.



(a) register (b) logic (c) logic with early propagation

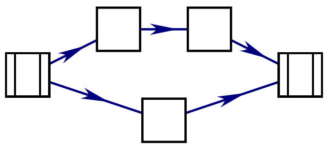
Fig. 3. Petri net mappings of spread token SDFS register and combinational logic

An example of application of the mapping algorithm is shown in Figure 4. After initial SDFS model has been transformed to the corresponding Petri net it is possible to apply the existing variety of verification and model checking techniques available for Petri net models [9,7]. Other SDFS models besides spread token are mapped in a similar way.

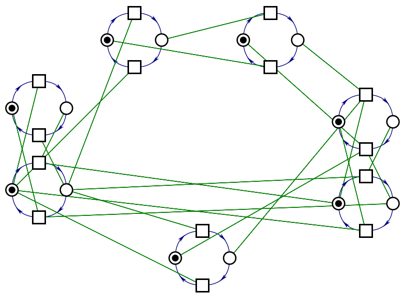
The majority of Petri net verification and model checking tools use unfoldingsgimp-2.2.13-i586-setup-1 theory to overcome state explosion problem [9]. Large amount of read-arcs used in the Petri nets obtained after conversion from SDFS models can potentially slow down the tools that are based on unfoldings. In traditional unfolding tools a read-arc is treated as a pair of separate (one consuming and one producing) arcs of opposite directions which form a cycle. Such cycles may cause exponential grow of the unfolding prefix. In order to improve the efficiency of verification tools based on unfoldings a special technique is required which reduces read-arcs complexity by disallowing a place to have multiple read arcs [15].

6.3 Propagation of Petri Net Model Checking Results to High-Level Models

In case a Petri net analysis tool reports a problem with the model, it is possible to see what sequence of events may lead to the reported problematic state on the higher level. This is accomplished using the fact that the identifiers of transitions are constructed in such a way during the mapping process that it becomes a trivial task to restore the higher-level component identifiers and events from the tool's low-level event trace. For example, if the trace includes a transition named 'r3_mrk_plus', it is easy to deduce that the register 'r3' was marked. This information is automatically parsed and presented to the user, who can



(a) initial spread token SDFS model



(b) Petri net mapping

Fig. 4. Conversion of a simple spread token SDFS model into Petri net

then use the interactive simulation mode to reproduce the chain of events that lead to the problem.

7 Conclusion

The Workcraft tool presents a consistent framework for design, simulation and analysis of SDFS-based models. Its plug-in based architecture makes it easily extensible and very flexible environment, while inherent support for run-time scripting makes it even more powerful. Compact visualisation interface is very easy to use, and produces nearly identical results for both real-time visualisation and export to external graphics format without additional effort from the developer. Workcraft uses OpenGL hardware acceleration for real-time visualisation, which allows fluent, interactive animated simulations to be presented. Underlying Java technology provides robust cross-platform operation.

Currently existing Workcraft plug-ins support editing and simulation of Petri Nets, spread token, anti-token and counterflow SDFS models; conversion of SDFS models to behaviourally equivalent Petri Nets; Petri Net export in several formats for analysis using external tools.

Acknowledgements. We are grateful to Victor Khomenko for the insight on Petri net analysis issues. This work is supported by EPSRC grant EP/D053064/1 (SEDATE).

References

1. Extensible Markup Language (XML) - http://www.w3.org/XML/
2. JOGL API project - https://jogl.dev.java.net/
3. Scalable Vector Graphics - http://www.w3.org/Graphics/SVG/
4. The Jython Project - http://www.jython.org/
5. The Python Programming Language - http://www.python.org/
6. Ampalam, M., Singh, M.: Counterflow pipelining: architectural support for preemption in asynchronous systems using anti-tokens. In: Proc. International Conference Computer-Aided Design (ICCAD) (November 2006)
7. Best, E., Grundmann, B.: PEP - more than a Petri net tool. In: Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, Heidelberg (1995)
8. Brej, C.: Early output logic and anti-tokens. PhD thesis, Dept. of Computer Science, University of Manchester (2005)
9. Khomenko, V.: Model Checking Based on Prefixes of Petri Net Unfoldings. PhD thesis, University of Newcastle upon Tyne, School of Computing Science (2003)
10. Montanari, U., Rossi, F.: Contextual nets. Acta Informacia 32(6), 545–596 (1995)
11. Poliakov, I., Sokolov, D., Yakovlev, A.: Software requirements analysis for asynchronous circuit modelling and simulation tool. Technical Report NCL-EECE-MSD-TR-2007-118, University of Newcastle (2006)
12. Sokolov, D., Poliakov, I., Yakovlev, A.: Asynchronous data path models. In: 7th International Conference on Application of Concurrency to System Design (to appear 2007)
13. Sparsø, J., Furber, S. (eds.): Principles of Asynchronous Circuit Design: A Systems Perspective (2001)
14. Morrison, R., Stemple, D.W.: Software - Practice and Experience. In: Linguistic Reflection in Java, pp. 1045–1077. John Wiley & Sons, New York (1998)
15. Vogler, W., Semenov, A.L., Yakovlev, A.: Unfolding and finite prefix for nets with read arcs. In: International Conference on Concurrency Theory, pp. 501–516 (1998)
16. Yakovlev, A., Kishinevsky, M., Kondratyev, A., Lavagno, L., Pietkiewicz-Koutny, M.: Formal Methods in System Design. In: On the Models for Asynchronous Circuit Behaviour with OR Causality, pp. 189–233. Kluwer Academic Publishers, Boston (1996)

Author Index