Lecture Notes in Computer Science 2360

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Javier Esparza Charles Lakos (Eds.)

Application and Theory of Petri Nets 2002

23rd International Conference, ICATPN 2002
Adelaide, Australia, June 24-30, 2002
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Javier Esparza
University of Edinburgh, Division of Informatics
James Clerk Maxwell Building, The King's Buildings
Mayfield Road, Edinburgh EH9 3JZ, United Kingdom
E-mail: jav@dcs.ed.ac.uk

Charles Lakos
University of Adelaide, Department of Computer Science
North Terrace, Adelaide, SA, 5005, Australia
E-mail: Charles.Lakos@adelaide.edu.au

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Application and theory of Petri nets 2002 : 23rd international conference ;
proceedings / ICATPN 2002, Adelaide, Australia, June 24 - 30, 2002.
Javier Esparza ; Charles Lakos (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2360)
ISBN 3-540-43787-8

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

ISSN 0302-9743
ISBN 3-540-43787-8 Springer-Verlag Berlin Heidelberg New York

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

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN 10870245 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 23rd International Conference on Application and Theory of Petri Nets. The aim of the Petri net conferences is to create a forum for discussing progress in the application and theory of Petri nets. Typically, the conferences have 100-150 participants – one third of these coming from industry while the rest are from universities and research institutions. The conferences always take place in the last week of June.

The conference and a number of other activities are co-ordinated by a steering committee with the following members: G. Balbo (Italy), J. Billington (Australia), G. De Michelis (Italy), S. Haddad (France), K. Jensen (Denmark), S. Kumagai (Japan), T. Murata (USA), C.A. Petri (Germany; honorary member), W. Reisig (Germany), G. Rozenberg (The Netherlands; chairman), and M. Silva (Spain).

Other activities before and during the 2002 conference included tool demonstrations, extensive introductory tutorials, two advanced tutorials on "Workflow Management: Models, Methods, and Systems" and "Model Checking", and two workshops on "Software Engineering and Formal Methods" and "Formal Methods Applied to Defence Systems". The tutorial notes and workshop proceedings are not published in this volume, but copies are available from the organizers. The proceedings can be found at `http://www.jrpit.flinders.edu.au/CRPIT.html`.

The 2002 conference was organized by the Computer Systems Engineering Centre, School of Electrical and Information Engineering at the University of South Australia, Adelaide, Australia with assistance from the Department of Computer Science, Adelaide University, Adelaide, Australia. We would like to thank the members of the organizing committee (see next page) and their teams.

We would like to thank very much all those who submitted papers to the Petri net conference. We received a total of 45 submissions from 18 different countries. This volume comprises the papers that were accepted for presentation. Invited lectures were given by W. van der Aalst, J. Desel, I. Hayes, C. Lakos, P.S. Thiagarajan, and A. Yakovlev (whose papers are included in this volume).

The submitted papers were evaluated by a program committee. The program committee meeting took place in Edinburgh, Scotland. We would like to express our gratitude to the members of the program committee, and to all the referees who assisted them. The names of these are listed on the following pages.

We would like to acknowledge the local support of Marco Kick, Bill Orrok, and Claus Schröter. Finally, we would like to mention the excellent co-operation with Springer-Verlag during the preparation of this volume.

April 2002 Javier Esparza and Charles Lakos

Organizing Committee

Carolyn Bellamy
Jonathan Billington (chair)
Pierre Dauchy
Guy Gallasch
Bing Han
Robin King
Lars Michael Kristensen
Charles Lakos
Lin Liu
Pauline Olsson
Patrick O'Sullivan
Chun Ouyang
Laure Petrucci

Tools Demonstration

Lars Michael Kristensen (chair)

Program Committee

Wil van der Aalst (The Netherlands)
Gianfranco Balbo (Italy)
Eike Best (Germany)
Jonathan Billington (Australia)
Gianfranco Ciardo (USA)
Jordi Cortadella (Spain)
Philippe Darondeau (France)
Giorgio De Michelis (Italy)
Javier Esparza (UK; co-chair; theory)
Claude Girault (France)
Nisse Husberg (Finland)
Lars Michael Kristensen (Denmark)
Sadatoshi Kumagai (Japan)
Charles Lakos (Australia; co-chair; applications)
Madhavan Mukund (India)
Wojciech Penczek (Poland)
Vladimiro Sassone (UK/Italy)
Karsten Schmidt (Germany)
Sol Shatz (USA)
Enrique Teruel (Spain)
Alex Yakovlev (UK)
Wlodek Zuberek (Canada)

Referees

Alessandra Agostini
Adrianna Alexander
Gerard Berthelot
Eric Badouel
Marek A. Bednarczyk
Simona Bernardi
Luca Bernardinello
Andrzej M. Borzyszkowski
Roberto Bruni
Benoît Caillaud
Felice Cardone
Søren Christensen
Paolo Ciancarini
Robert Clariso
José-Manuel Colom
Deepak D'Souza
Michel Diaz
Claude Dutheillet
Susanna Donatelli
Emmanuelle Encrenaz
Robert Esser
Joaquín Ezpeleta
David de Frutos-Escrig
Hans Fleischhack
Giuliana Franceschinis
Rossano Gaeta
Steven Gordon
Marco Gribaudo
Bing Han
Keijo Heljanko
Zhaoxia Hu
Guy Juanole
Jorge Júlvez
Jens Bæk Jørgensen
Victor Khomenko
Ekkart Kindler
Mike Kishinevsky
Hanna Klaudel
Maciej Koutny

K Narayan Kumar
Johan Lilius
Lin Liu
Louise Lorentsen
Thomas Mailund
Axel Martens
José Meseguer
Toshiyuki Miyamoto
Kjeld Høyer Mortensen
Alix Munier
Tadao Murata
Marko Mäkelä
Chun Ouyang
Emmanuel Paviot-Adet
Denis Poitrenaud
Giuseppe Pappalardo
Elina Parviainen
Enric Pastor
Olli-Matti Penttinen
Laure Petrucci
Agata Półrola
Lucia Pomello
Pierre-Olivier Ribet
Laura Recalde
Wolfgang Reisig
Elvinia Riccobene
Diego Rodríguez
Matteo Sereno
Manuel Silva
Radu I. Siminiceanu
Pawel Sobocinski
Jeremy Sproston
Jiří Srba
Peter Starke
Christian Stehno
K.V. Subrahmanyam
Ichiro Suzuki
Maciej Szreter
Cecile Bui Thanh
P.S. Thiagarajan
Fernando Tricas
Teemu Tynjälä
François Vernadat
Kimmo Varpaaniemi
Eric Verbeek
José-Luis Villarroel
Harro Wimmel
Jozef Winkowski
Bozena Wozna
Fei Xia
Xiaolan Xie
Haiping Xu
Hideki Yamasaki

Sponsoring Institutions

The Australian Defence Science and Technology Organization
The Adelaide City Council
The University of South Australia
Division of Informatics, University of Edinburgh

Table of Contents

Invited Papers

Making Work Flow: On the Application of Petri Nets to Business Process Management 1
Wil M. P. van der Aalst

Model Validation – A Theoretical Issue? 23
Jörg Desel

The Real-Time Refinement Calculus: A Foundation for Machine-Independent Real-Time Programming 44
Ian J. Hayes

The Challenge of Object Orientation for the Analysis of Concurrent Systems 59
Charles Lakos

Abstract Cyclic Communicating Processes: A Logical View 68
P. S. Thiagarajan

Is the Die Cast for the Token Game? 70
Alex Yakovlev

Regular Papers

Quasi-Static Scheduling of Independent Tasks for Reactive Systems 80
Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone, and Yosinori Watanabe

Data Decision Diagrams for Petri Net Analysis 101
Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier

Non-controllable Choice Robustness Expressing the Controllability of Workflow Processes 121
Juliane Dehnert

Real-Time Synchronised Petri Nets 142
Giovanna Di Marzo Serugendo, Dino Mandrioli, Didier Buchs, and Nicolas Guelfi

Computing a Finite Prefix of a Time Petri Net 163
Hans Fleischhack and Christian Stehno

Verification of a Revised WAP Wireless Transaction Protocol 182
Steven Gordon, Lars Michael Kristensen, and Jonathan Billington

Characterizing Liveness of Petri Nets in Terms of Siphons 203
Li Jiao, To-Yat Cheung, and Weiming Lu

Petri Nets, Situations, and Automata 217
Ekkart Kindler

Reproducibility of the Empty Marking 237
Kurt Lautenbach

Modeling and Analysis of Multi-class Threshold-Based Queues with Hysteresis Using Stochastic Petri Nets 254
Louis-Marie Le Ny and Bruno Tuffin

Tackling the Infinite State Space of a Multimedia Control Protocol Service Specification 273
Lin Liu and Jonathan Billington

Modelling of Features and Feature Interactions in Nokia Mobile Phones Using Coloured Petri Nets 294
Louise Lorentsen, Antti-Pekka Tuovinen, and Jianli Xu

Analysing Infinite-State Systems by Combining Equivalence Reduction and the Sweep-Line Method 314
Thomas Mailund

Regular Event Structures and Finite Petri Nets: The Conflict-Free Case ... 335
Mogens Nielsen and P. S. Thiagarajan

A Formal Service Specification for the Internet Open Trading Protocol 352
Chun Ouyang, Lars Michael Kristensen, and Jonathan Billington

Transition Refinement for Deriving a Distributed Minimum Weight Spanning Tree Algorithm 374
Sibylle Peuker

Token-Controlled Place Refinement in Hierarchical Petri Nets with Application to Active Document Workflow 394
David G. Stork and Rob van Glabbeek

Translating TPAL Specifications into Timed-Arc Petri Nets 414
Valentín Valero, Juan José Pardo, and Fernando Cuartero

Tool Presentation

Maria: Modular Reachability Analyser for Algebraic System Nets 434
Marko Mäkelä

Author Index 445

Making Work Flow: On the Application of Petri Nets to Business Process Management

Wil M. P. van der Aalst ⋆

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl

Abstract. Information technology has changed business processes within and between enterprises. More and more work processes are being conducted under the supervision of information systems that are driven by process models. Examples are workflow management systems such as Staffware, enterprise resource planning systems such as SAP and Baan, but also include many domain specific systems. It is hard to imagine enterprise information systems that are unaware of the processes taking place. Although the topic of business process management using information technology has been addressed by consultants and software developers in depth, a more fundamental approach has been missing. Only since the nineties, researchers started to work on the foundations of business process management systems. This paper addresses some of the scientific challenges in business process management. In the spirit of Hilbert's problems[1], 10 interesting problems for people working on Petri-net theory are posed.

1 Introduction

The goal of this paper is to show the relevance, architecture, and Achilles heel of business process management systems. This way we hope to interest Petri-net researchers in some of the scientific challenges in this domain. The definition of a business process management system used throughout this paper is: *a generic software system that is driven by explicit process designs to enact and manage operational business processes*. The system should be process-aware and generic in the sense that it is possible to modify the processes it supports. The process designs are often graphical and the focus is on structured processes that need to handle many cases.

In the remainder of this paper, we will first put business process management and related technology in its historical context. Then, we will discuss models for process design. Since business process management systems are driven by explicit models, it is important to use the right techniques. Next, we will discuss

⋆ Part of paper is taken from my inaugural lecture "Making Work Flow: On the Design, Analysis, and Enactment of Business Processes" [6].

[1] Note that by no means we are suggesting that the problems in this paper are of the same stature as the 23 problems raised by David Hilbert in 1900.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 1–22, 2002.
© Springer-Verlag Berlin Heidelberg 2002

techniques for the analysis of process models. We will argue that it is vital to have techniques to assert the correctness of workflow designs. Based on this we will focus on systems for process enactment, i.e., systems that actually make the "work flow" based on a model of the processes and organizations involved. Finally, we will pose 10 interesting problems in the spirit of Hilbert's problems [31].

2 Business Process Management from a Historical Perspective

Only the wisest and stupidest of men never change.
Confucius

To show the relevance of business process management systems, it is interesting to put them in a historical perspective. Consider Figure 1, which shows some of the ongoing trends in information systems. This figure shows that today's information systems consist of a number of layers. The center is formed by the operating system, i.e., the software that makes the hardware work. The second layer consists of generic applications that can be used in a wide range of enterprises. Moreover, these applications are typically used within multiple departments within the same enterprise. Examples of such generic applications are a database management system, a text editor, and a spreadsheet program. The third layer consists of domain specific applications. These applications are only used within specific types of enterprises and departments. Examples are decision support systems for vehicle routing, call center software, and human resource management software. The fourth layer consists of tailor-made applications. These applications are developed for specific organizations.

In the sixties the second and third layer were missing. Information systems were built on top of a small operating system with limited functionality. Since no generic nor domain specific software was available, these systems mainly consisted of tailor-made applications. Since then, the second and third layer have developed and the ongoing trend is that the four circles are increasing in size, i.e., they are moving to the outside while absorbing new functionality. Today's operating systems offer much more functionality. Database management systems that reside in the second layer offer functionality which used to be in tailor-made applications. As a result of this trend, the emphasis shifted from programming to assembling of complex software systems. The challenge no longer is the coding of individual modules but orchestrating and gluing together pieces of software from each of the four layers.

Another trend is the shift from data to processes. The seventies and eighties were dominated by data-driven approaches. The focus of information technology was on storing and retrieving information and as a result data modeling was the starting point for building an information system. The modeling of business processes was often neglected and processes had to adapt to information technology. Management trends such as business process reengineering illustrate the increased emphasis on processes. As a result, system engineers are resorting to a more process driven approach.

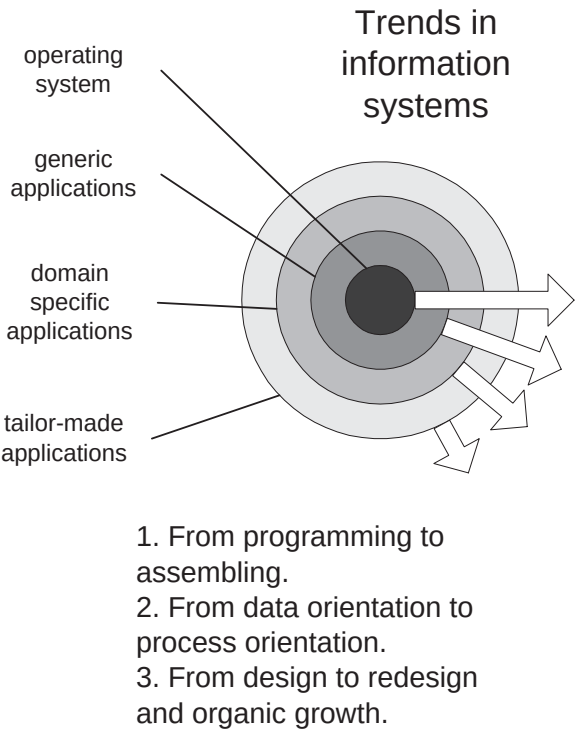


Fig. 1. Trends relevant for business process management

The last trend we would like to mention is the shift from carefully planned designs to redesign and organic growth. Due to the omnipresence of the Internet and its standards, information systems change on-the-fly. As a result, fewer systems are built from scratch. In many cases existing applications are partly used in the new system. Although component-based software development still has it problems, the goal is clear and it is easy to see that software development has become more dynamic.

The trends shown in Figure 1 provide a historical context for business process management systems. Business process management systems are either separate applications residing in the second layer or are integrated components in the domain specific applications, i.e., the third layer. Notable examples of business process management systems residing in the second layer are workflow management systems [33,36] such as Staffware, MQSeries, and COSA, and case handling systems such as FLOWer. Note that leading enterprise resource planning systems populating the third layer also offer a workflow management module. The workflow engines of SAP, Baan, PeopleSoft, Oracle, and JD Edwards can be considered as integrated business process management systems. The idea to isolate the management of business processes in a separate component is consis-

tent with the three trends identified. Business process management systems can be used to avoid hard-coding the work processes into tailor-made applications and thus support the shift from programming to assembling. Moreover, process orientation, redesign, and organic growth are supported. For example, today's workflow management systems can be used to integrate existing applications and support process change by merely changing the workflow diagram. Give these observations, we hope to have demonstrated the practical relevance of business process management systems. In the remainder of this paper we will focus more on the scientific importance of these systems. Moreover, for clarity we will often restrict the discussion to clear cut business process management systems such as workflow management systems.

An interesting starting point from a scientific perspective is the early work on office information systems. In the seventies, people like Skip Ellis [24], Anatol Holt [32], and Michael Zisman [46] already worked on so-called office information systems, which were driven by explicit process models. It is interesting to see that the three pioneers in this area independently used Petri-net variants to model office procedures. During the seventies and eighties there was great optimism about the applicability of office information systems. Unfortunately, few applications succeeded. As a result of these experiences, both the application of this technology and research almost stopped for a decade. Consequently, hardly any advances were made in the eighties. In the nineties, there again was a huge interest in these systems. The number of workflow management systems developed in the past decade and the many papers on workflow technology illustrate the revival of office information systems. Today workflow management systems are readily available [36]. However, their application is still limited to specific industries such as banking and insurance. As was indicated by Skip Ellis it is important to learn from these ups and downs [26]. The failures in the eighties can be explained by both technical and conceptual problems. In the eighties, networks were slow or not present at all, there were no suitable graphical interfaces, and proper development software was missing. However, there were also more fundamental problems: a unified way of modeling processes was missing and the systems were too rigid to be used by people in the workplace. Most of the technical problems have been resolved by now. However, the more conceptual problems remain. Good standards for business process modeling are still missing and even today's workflow management systems enforce unnecessary constrains on the process logic (e.g., processes are made more sequential).

One of the great challenges of business process management systems is to offer both support and flexibility [9,14,35]. Today's systems typically are too rigid, thus forcing people to work around the system. One of the problems is that software developers and computer scientists are typically inspired by processes inside a computer system rather than processes outside a computer. Figure 2 illustrates the typical mind-frame of people developing business process management systems. This photograph shows the Whirlwind computer, which was the first computer system to have magnetic core memory (1953). It is interesting to mention that Whirlwind was developed by Jay Forrester who also

WHIRLWIND (1953)

Architecture: 32 bit word length, duplex CPU, 75kips single address, no interrupts, 4 index registers, real time clock
Memory: magnetic core (4Kx64word) 6 microseconds cycle time; magnetic drum (150K word); 4 IBM Model 729 tape drives (~100K word each); parity checking
I/O: CRT display, keyboard, light gun, real time serial data (teletype 1300bps modem), voice line
Size: 60,000 vacuum tubes, 175,000 diodes, 13,000 transistors; CPU space 50x150 feet each; CPU weight 500,000 lbs; power consumption: 3 megawatts

Fig. 2. The Whirlwind - photo from the Timeline of Events on Computer History (©2001 IEEE)

developed the well-known Systems Dynamics approach [27]. Software engineers are typically trained in the architecture and systems software of computers like the Whirlwind and its successors. As a result, these engineers think in terms of control systems rather than support systems. This explains that few of the existing workflow management systems allow for the so-called implicit choice, i.e., a choice resolved by the environment rather than the system [13]. To solve these problems, the typical mind-frame should be changed such that the business process management system is viewed as a reactive system rather than merely a control system.

To summarize we state that, although the relevance of business process management systems is undisputed, many fundamental problems remain to be solved. In the remainder of this paper we will try to shed light on some of these problems.

3 Models for Process Design

A camel is a horse designed by committee.
Sir Alec Issigonis

Business process management systems are driven by models of processes and organizations. By changing these models, the behavior of the system adapts to its environment and changing requirements. These models cover different perspectives. Figure 3 shows some of the perspectives relevant for business process management systems [33]. The process perspective describes the control-flow, i.e., the ordering of tasks. The information perspective describes the data that are used. The resource perspective describes the structure of the organization and identifies resources, roles, and groups. The task perspective describes the

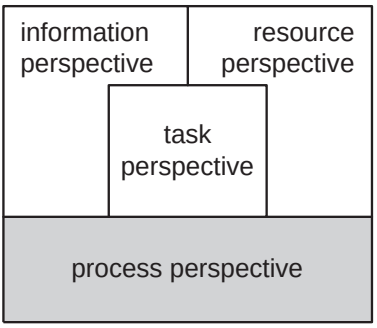


Fig. 3. Perspectives of models driving business process management systems

content of individual steps in the processes. Each perspective is relevant. However, the process perspective is dominant for the type of systems addressed in this talk.

Many techniques have been proposed to model the process perspective. Some of these techniques are informal in the sense that the diagrams used have no formally defined semantics. These models are typically very intuitive and the interpretation shifts depending on the modeler, application domain, and characteristics of the business processes at hand. Examples of informal techniques are ISAC, DFD, SADT, and IDEF. These techniques may serve well for discussing work processes. However, they are inadequate for directly driving information systems since they are incomplete and subject to multiple interpretations. Therefore, more precise ways of modeling are required.

Figure 4 shows an example of an order handling process modeled in terms of a so-called workflow net [1]. Workflow nets are based on the classical Petri-net model invented by Carl Adam Petri in the early sixties [39]. The squares are the active parts of the model and correspond to tasks. The circles are the passive parts of the model and are used to represent states. In the classical Petri net, the squares are named transitions and the circles places. A workflow net models the life-cycle of one case. Examples of cases are insurance claims, tax declarations, and traffic violations. Cases are represented by tokens and in this case the token in *start* corresponds to an order. Task *register* is a so-called AND-split and is enabled in the state shown. The arrow indicates that this task requires human intervention. If a person executes this task, the token is removed from place *start* and two tokens are produced: one for *c1* and one for *c2*. Then, in parallel, two tasks are enabled: *check_availability* and *send_bill*. Depending on the eagerness of the workers executing these two tasks either *check_available* or *send_bill* is executed first. Suppose *check_availability* is executed first. If the ordered goods are available, they can be shipped by executing task *ship_goods*. If they are not available, either a replenishment order is issued or not. Note that *check_availability* is an OR-split and produces one token for *c3*, *c4*, or *c5*. Suppose that not all ordered goods are available, but the appropriate replenishment orders were already issued. A token is produced for *c3* and task *update* becomes enabled.

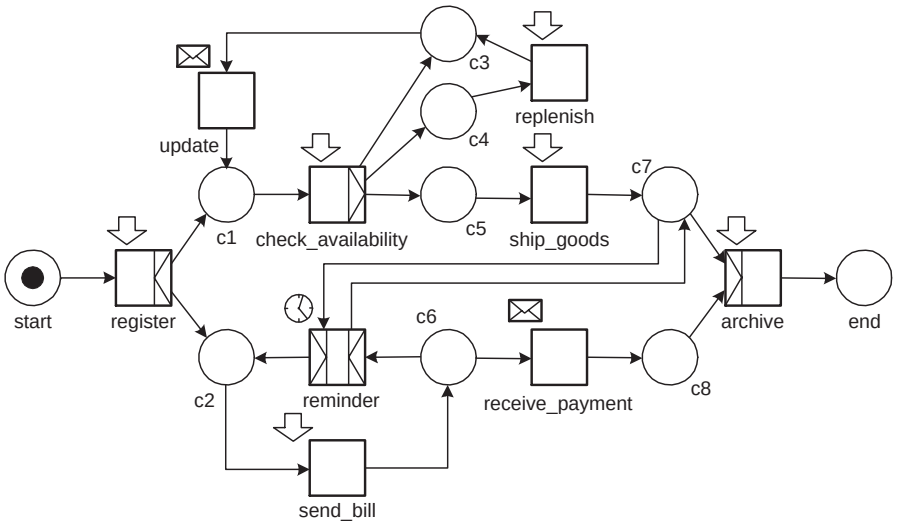


Fig. 4. WF-net

Suppose that at this point in time task *send_bill* is executed, resulting in the state with a token in *c3* and *c6*. The token in *c6* is input for two tasks. However, only one of these tasks can be executed and in this state only *receive_payment* is enabled. Task *receive_payment* can be executed the moment the payment is received. Task *reminder* is an AND-join/AND-split and is blocked until the bill is sent and the goods have been shipped. Note that the reminder is sent after a specified period as indicated by the clock symbol. However, it is only possible to send a remainder if the goods have been actually shipped. Assume that in the state with a token in *c3* and *c6* task *update* is executed. This task does not require human involvement and is triggered by a message of the warehouse indicating that relevant goods have arrived. Again *check_availability* is enabled. Suppose that this task is executed and the result is positive. In the resulting state *ship_goods* can be executed. Now there is a token in *c6* and *c7* thus enabling task *reminder*. Executing task *reminder* again enables the task *send_bill*. A new copy of the bill is sent with the appropriate text. It is possible to send several reminders by alternating *reminder* and *send_bill*. However, let us assume that after the first loop the customer pays resulting in a state with a token in *c7* and *c8*. In this state, the AND-join *archive* is enabled and executing this task results in the final state with a token in *end*.

This very simple workflow net shows some of the routing constructs relevant for business process modeling. Sequential, parallel, conditional, and iterative routing are present in this model. There also are more advanced constructs such as the choice between *receive_payment* and *reminder*. This is a so-called *implicit choice* since it is not resolved by the system but by the environment of the system. The moment the bill is sent, it is undetermined whether *receive_payment* or

reminder will be the next step in the process. Another advanced construct is the fact that task *reminder* is blocked until the goods have been shipped. The latter construct is a so-called *milestone*. The reason that we point out both constructs is that many systems have problems supporting these rather fundamental process patterns [13].

Workflow nets have clear semantics. The fact that we are able to play the so-called token game using a minimal set of rules shows the fact that these models are executable. None of the informal informal techniques mentioned before (i.e., ISAC, DFD, SADT, and IDEF) have formal semantics. Besides workflow nets there are many other formal techniques. Examples are the many variants of process algebra [17] and statecharts [29]. The reason we prefer to use a variant of Petri nets is threefold [1]:

- Petri nets are graphical and yet precise.
- Petri nets offer an abundance of analysis techniques.
- Petri nets treat states as first-class citizens.

The latter point deserves some more explanation. Many techniques for business process modeling focus exclusively on the active parts of the process, i.e., the tasks. This is rather surprising since in many administrative processes the actual processing time is measured in minutes and the flow time is measured in days. This means that most of the time cases are in-between two subsequent tasks. Therefore, it is vital to model these states explicitly.

In recent years, the Unified Modeling Language (UML, [20]) has become the de facto standard for software development. UML has four diagrams for process modeling. UML supports variants of statecharts and its activity diagrams are inspired by Petri nets. UML combines both good and bad ideas and can be considered semi-formal. Many colleagues are trying to provide solid semantics for UML. In my opinion, it would have been better to start with a solid foundation.

4 Techniques for Process Analysis

From the errors of others, a wise man corrects his own.
Syrus

Business process management systems allow organizations to change their processes by merely changing the models. The models are typically graphical and can be changed quite easily. This provides more flexibility than conventional information systems. However, by reducing the threshold for change, errors are introduced more easily. Therefore, it is important to develop suitable analysis techniques. However, it is not sufficient to just develop these techniques. It is as least as important to look at methods and tools to make them applicable in a practical context.

Traditionally, most techniques used for the analysis of business processes, originate from operations research. All students taking courses in operations management will learn to apply techniques such as simulation, queueing theory,

and Markovian analysis. The focus mainly is on *performance analysis* and less attention is paid to the correctness of models. *Verification* and *validation* are often neglected. As a result, systems fail by not providing the right support or even break down [2,42]. Verification is needed to check whether the resulting system is free of logical errors. Many process designs suffer from deadlocks and livelocks that could have been detected using verification techniques. Validation is needed to check whether the system actually behaves as expected. Note that validation is context dependent while verification is not. A system that deadlocks is not correct in any situation. Therefore, verifying whether a system exhibits deadlocks is context independent. Validation is context dependent and can only be done with knowledge of the intended business process.

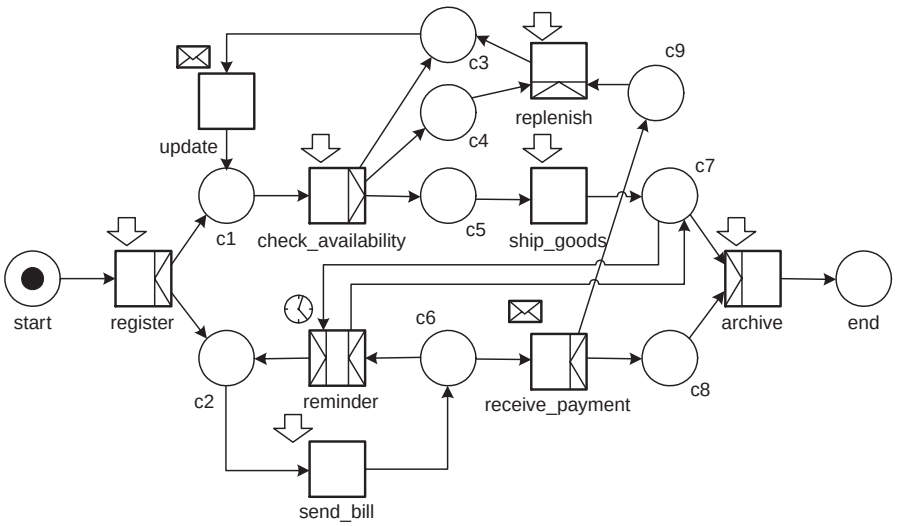


Fig. 5. An incorrect WF-net

To illustrate the relevance of validation and verification and to demonstrate some of the techniques available, we return to the workflow net shown in Figure 4. This workflow process allows for the situation where a replenishment is issued before any payment is received. Suppose that we want to change the design such that replenishments are delayed until receiving payment. An obvious way to model this is to connect task *receive_payment* with *replenish* using an additional place *c9* as shown in Figure 5. Although this extension seems to be correct at first glance, the resulting workflow net has several errors. The workflow will deadlock if a second replenishment is needed and something is left behind in the process if no replenishments are needed. These are logical errors that can be detected without any knowledge of the order handling process. For verification, application independent notions of correctness are needed. One of these notions is the so-called *soundness property* [1]. A workflow net is sound if and only if the

workflow contains no dead parts (i.e., tasks that can never be executed), from any reachable state it is always possible to terminate, and the moment the workflow terminates all places except the sink place (i.e., place *end*) are empty. Note that soundness rules out logical errors such as deadlocks and livelocks. The notion of soundness is applicable to any workflow language. An interesting observation is that soundness corresponds to liveness and boundedness of the short-circuited net [1]. The latter properties have been studied extensively [41,23]. As a result, powerful analysis techniques and tools can be applied to verify the correctness of a workflow design. Practical experience shows that many errors can be detected by verifying the soundness property. Moreover, Petri-net theory can also be applied to guide the designer towards the error.

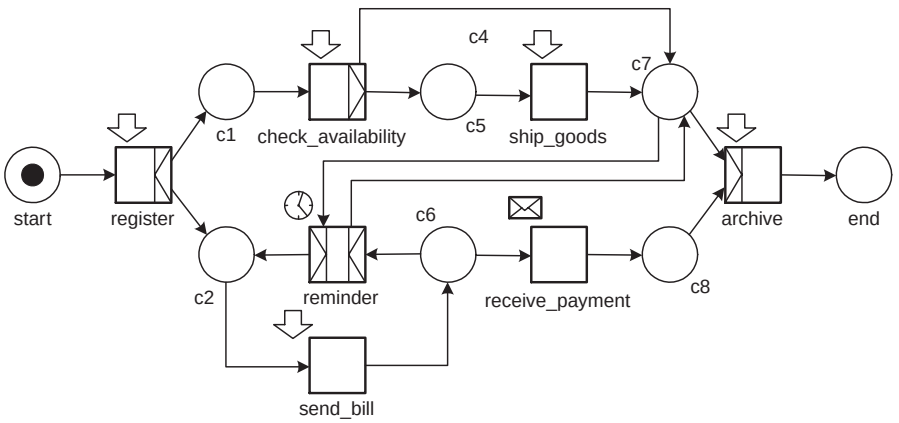


Fig. 6. A sound but incorrect WF-net

Soundness does not guarantee that the workflow net behaves as intended. Consider for example, the workflow net shown in Figure 6. Compared to the original model, the shipment of goods is skipped if some of the goods are not available. Again this may seem to be a good idea at first glance. However, customers are expected to pay even if the goods are never delivered. In other words, task *receive_payment* needs to be executed although task *ship_goods* may never be executed. The latter error can only be detected using knowledge about the context. Based on this context one may decide whether this is acceptable or not. Few analysis techniques exist to automatically support this kind of validation. The only means of validation offered by today's workflow management systems is gaming and simulation.

An interesting technique to support validation is inheritance of dynamic behavior. Inheritance can be used as a technique to compare processes. Inheritance relates subclasses with superclasses [19]. A workflow net is a subclass of a superclass workflow net if certain dynamic properties are preserved. A subclass

typically contains more tasks. If by hiding and/or blocking tasks in the subclass one obtains the superclass, the subclass inherits the dynamics of the superclass.[2] The superclass can be used to specify the minimal properties the workflow design should satisfy. By merely checking whether the actual design is a subclass of the superclass, one can validate the essential properties. Consider for example Figure 7. This workflow net describes the minimal requirements the order handling process should satisfy. The tasks *register*, *ship_goods*, *receive_payment*, and *archive* are mandatory. Tasks *ship_goods* and *receive_payment* may be executed in parallel but should be preceded by *register* and followed by *archive*. The original order handling process shown in Figure 4 is a subclass of this superclass. Therefore, the minimal requirements are satisfied. However, the order handling process shown in Figure 6 is not a subclass. The fact that task *ship_goods* can be skipped demonstrates that not all properties are preserved.

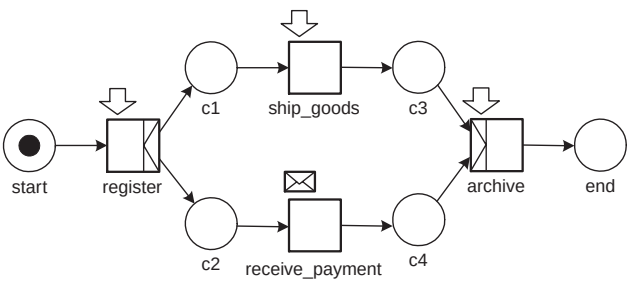


Fig. 7. A superclass WF-net

Inheritance of dynamic behavior is a very powerful concept that has many applications. Inheritance-preserving transformation rules and transfer rules offer support at design-time and at run-time [8]. Subclass-superclass relationships also can be used to enforce correct processes in an E-commerce setting. If business partners only execute subclass processes of some common contract process, then the overall workflow will be executed as agreed. It should be noted that workflows crossing the borders of organizations are particularly challenging from a verification and validation point of view [4]. Errors resulting from miscommunication between business partners are highly disruptive and costly. Therefore, it is important to develop techniques and tools for the verification and validation of these processes.

Few tools aiming at the verification of workflow processes exist. Woflan [43] and Flowmake [42] are two notable exceptions. We have been working on Woflan since 1997. Figure 8 shows a screenshot of Woflan. Woflan combines state-of-the-art scientific results with practical applications [3,11,43,45]. Woflan can interface with leading workflow management systems such as Staffware and COSA. It can

[2] We have identified four notions of inheritance. In this paper, we only refer to life-cycle inheritance.

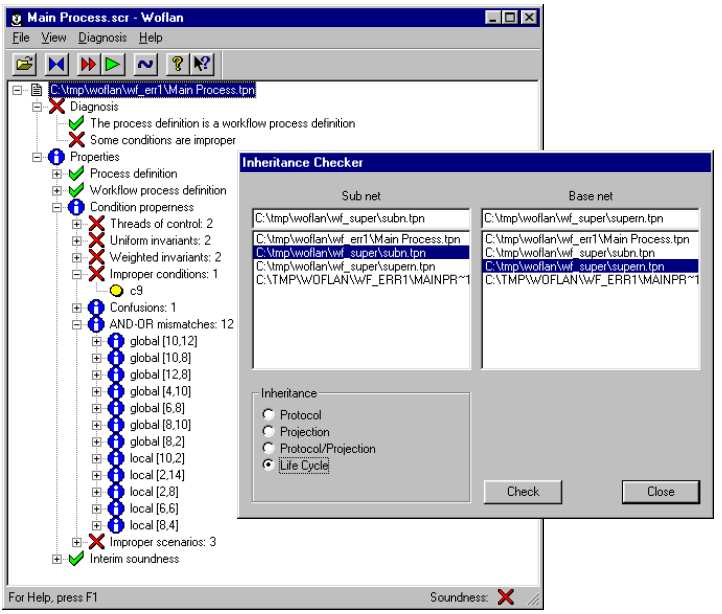


Fig. 8. A screenshot showing the verification and validation capabilities of Woflan

also interface with BPR-tools such as Protos. Workflow processes designed using any of these tools can be verified for correctness. It turns out that the challenge is not to decide whether the design is sound or not. The real challenge is to provide diagnostic information that guides the designer to the error. Woflan also supports the inheritance notions mentioned before. Given two workflow designs, Woflan is able to decide whether one is a subclass of the other. Tools such as Woflan illustrate the benefits of a more fundamental approach. Large scale experiments with experienced students show that workflow designers frequently make errors and that these design errors can be detected using Woflan [43].

5 Systems for Process Enactment

> *If the automobile had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get one million miles to the gallon, and explode once a year, killing everyone inside.*
>
> **Robert Cringely**

Progress in computer hardware has been incredible. In 1964 Gordon Moore, predicted that the number of elements on a produced chip would double every 18 months.[3] Up until now, Moore's law still applies. Information technology has also

[3] Moore (founder of Intel), commenting on the growth of the microelectronics industry in 1964, noted a doubling of the number of elements on a produced chip once every 12

resulted in a spectacular growth of the information being gathered. The commonly used term "information overload" illustrates this growth. It is estimated that for each individual, i.e., child, man, and woman, 250 megabytes of data are gathered each year [37]. The Internet and the World-Wide-Web have made an abundance of information available at low costs. However, despite the apparent progress in computer hardware and information processing, many information systems leave much to be desired. Typically, software contains errors and people need to work around the system to get things done. These observations justify the use of solid models and analysis techniques, as discussed before.

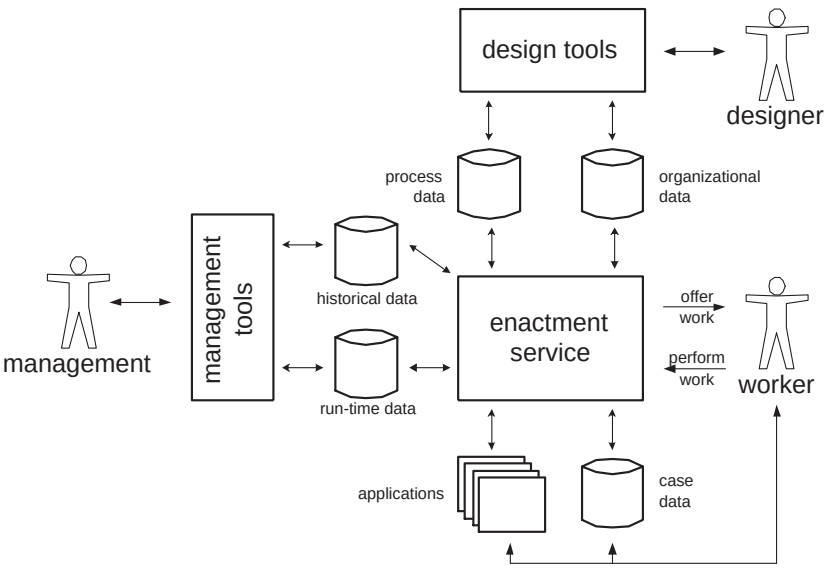


Fig. 9. The architecture of a business process management system

Thus far, the focus of this paper has been on the design and analysis of work processes. Now it is time to focus on the systems to enact these work processes. Figure 9 shows the typical architecture of a business process management system. The designer uses the design tools to create models describing the processes and the structure of the organization. The manager uses management tools to monitor the flow of work and act if necessary. The worker interacts with the enactment service. The enactment service can offer work to workers and workers can search, select and perform work. To support the execution of tasks, the enactment service may launch various kinds of applications. Note that the enactment service is the core of the system deciding on "what", "how", "when", and

months. For a decade that meant a growth factor of approximately 1000. Today, when Moore's Law is quoted, the time constant typically quoted is 18 months. However, some argue that a constant of 24 months is more appropriate.

"by whom". Clearly, the enactment service is driven by models of the processes and the organizations. By merely changing these models the system evolves and adapts. This is the ultimate promise of business process management systems.

Today's workflow management systems have an architecture consistent with Figure 9. Consider, for example, the screenshots of Staffware shown in Figure 10. Staffware is one of the leading workflow management systems. The top window shows the design tool of Staffware while defining a simple workflow process. Work is offered through so-called work queues. One worker can have multiple work queues and one work queue can be shared among multiple workers. The window in the middle shows the set of available work queues (left) and the content of one of these work queues (right). The bottom window shows an audit trail of a case. The three windows show only some of the capabilities offered by contemporary workflow management systems. It is fairly straightforward to map these windows onto the architecture. In other processes-aware information systems such as enterprise resource planning systems, one will find the architecture shown in Figure 9 embedded in a larger architecture.

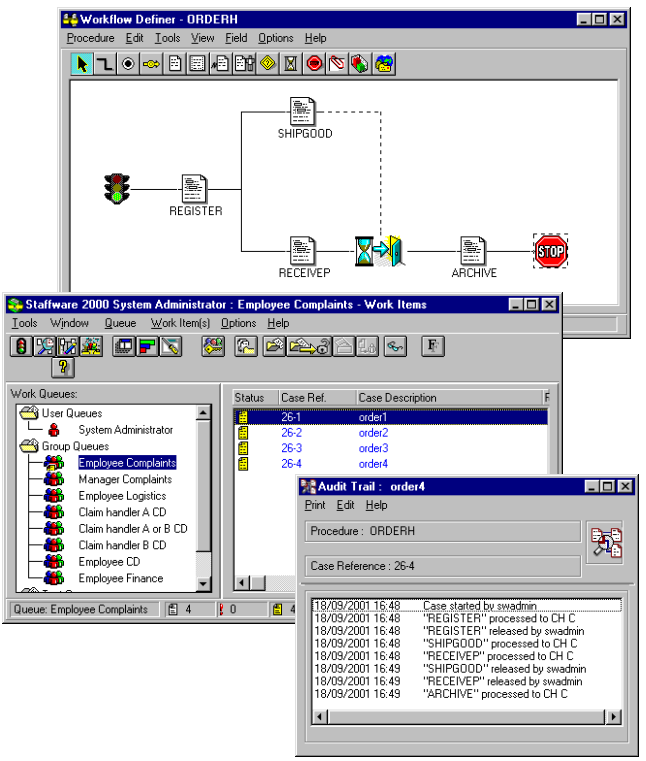


Fig. 10. The Graphical Workflow Definer, Work Queue, and Audit Trail of Staffware

Despite the acceptance of process-aware information systems, the current generation of products leaves much to be desired. In the next section we will highlight some of the challenging problems in this domain.

6 Challenging Problems

> *A scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die and a new generation grows up that is familiar with it.*
> **Maxwell Planck**

In 1900 David Hilbert gave a lecture before the International Congress of Mathematicians in Paris. In this lecture he gave gave a massive homework assignment to all the mathematicians of the world in the form of 23 problems [31]. These problems became the backbone for mathematical research in the 20th century. Most of the problems have been partially solved; some have been restated and the new interpretations have been solved. By merely formulating a number of problems, Hilbert directed research efforts towards the problems he thought to be relevant. In this paper, we are trying to direct the attention of researchers to 10 Petri-net-related problems relevant for business process management.

1. *What is the complexity of deciding soundness?*
   Given a workflow net it is possible to decide soundness by constructing and inspecting the reachability graph. It is also shown that soundness coincides with liveness and boundedness of the short-circuited net [1]. For subclasses such as free-choice workflow nets and workflow nets without PT- and TP-handles soundness can be analyzed in polynomial time [5]. However, for arbitrary workflow nets, the best known algorithm is non-primitive recursive space [43]. An open question is whether the structure of a workflow net can be exploited to improve this algorithm.
2. *How to calculate differences/commonalities of processes?*
   Inheritance is well-defined for static structures such as classes, e.g., a class is a subclass of another class if it has the attributes/methods of the superclass. It is more difficult to compare processes, cf. [19] for a discussion on this topic. The absence of a well-established notion of inheritance makes it difficult to compare processes let alone reason about differences/commonalities of processes. For example, one can have two processes sharing the same set of tasks. This does not imply that one can be substituted by the other. Similarly, it is difficult to define what two processes have in common if they do not agree on the ordering of tasks. A first attempt to reason about differences/commonalities of processes was given in [7]. Unfortunately, existing notions are unsatisfactory since they provide only partial solutions and their computational complexity is unknown. Many practical questions are related to these issues. For example, when harmonizing procedures of different enterprises/govenments it is interesting to calculate what they have in common and list the differences. Therefore, the quest for good notions and algorithms to calculate differences/commonalities of processes is highly relevant.

3. *Which workflow process patterns are the essential ones?*
   The Workflow Management Coalition (WFMC) has standardized basic building blocks such as AND/XOR-splits/joins [36]. Using these building blocks one can model sequential, parallel, conditional, and iterative processes. However, these basic control-flow patterns are unable to directly cope with more advanced patterns. Some of these more advanced patterns are described in [12,13]. Examples are patterns dealing with multiple instances, e.g., an insurance claim with a variable number of witness statements, and advanced synchronization patterns, e.g., the n-out-of-m join and the OR-join (only synchronize when needed). Some patterns can be modeled quite easily in terms of Petri nets. Other patterns can only be modeled after introducing color sets and spaghetti-like diagrams. After four decades of Petri-net modeling, a mature and complete set of process patterns is still missing and designers are re-inventing the wheel every day. Clearly, most patterns are domain dependent. The set of patterns reported in [12,13] is specific for workflow management. The question is whether this set is complete. Another question is how frequently these patterns are used/needed in practice.
4. *How to evaluate and compare the expressive power of workflow management systems?*
   Over the last decade, more than 200 different commercial workflow management systems became available. Surprisingly, almost every system uses a propriety language for modeling workflow processes (despite the efforts of the WFMC [36]). Only a handful of systems is based on formal methods such as Petri nets (e.g., COSA by Thiel AG and Income by Promatis). Many systems use a vendor-specific variant of flow-charts extended with parallelism or Petri-net like diagrams such as the event-driven process chains (e.g., ARIS and SAP). Some systems also use very different principles such as false-token propagation (e.g., InConcert by TIBCO and MQSeries Workflow by IBM) and data-driven controls (e.g., FLOWer by Pallas Athena). This makes it very difficult to compare systems. Nevertheless, it is important to have insights in the expressive power of these systems. Note that the term "expressive power" is not used in the traditional sense. The term is used to refer to the effort needed to construct models that reflect the process logic in a direct manner. Note that the patterns mentioned before can be used as a stepping stone for developing a method for evaluating and comparing process-enabled systems such as workflow management systems.
5. *Which class of workflow processes can be rediscovered?*
   Contemporary workflow management systems are driven by explicit process models, i.e., a completely specified workflow design is required in order to enact a given workflow process. Creating a workflow design is a complicated time-consuming process and typically there are discrepancies between the actual workflow processes and the processes as perceived by the management, workers, and ultimately the workflow designer. Therefore, it is interesting to develop techniques for (re)discovering workflow models based on observations of the real workflow. Starting point for such techniques are so-called "workflow logs" containing information about the workflow process

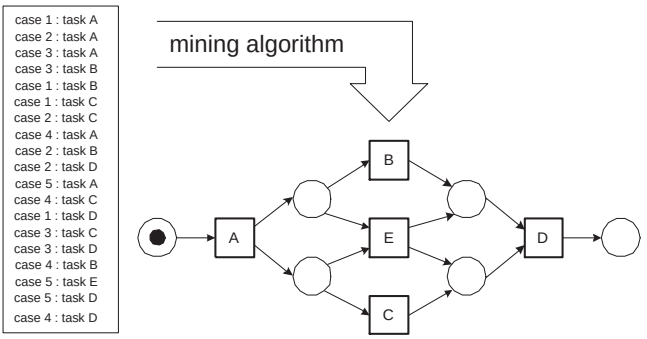


Fig. 11. Using mining techniques it is possible to distill a workflow net (right) from a workflow log (left)

as it is actually being executed (cf. Figure 11). Unfortunately, it is difficult to (re)discover an arbitrary workflow process. All known process mining techniques fail to properly detect non-free-choice structures [22,44]. An open question is which class of workflow processes (i.e., a subclass of workflow nets) can be rediscovered assuming "complete" logs.

6. *Which are the minimal requirements needed to ensure the correct cooperation between workflow processes?*
   The rise of E-business has increased the number of workflow processes crossing organizational boundaries. As a result, there is a need to connect autonomous workflow processes in such a way that some overall objective is satisfied. A minimal correctness criterion for a set of interoperating workflow nets is soundness. Suppose that each of the local workflow nets is sound. Which requirements should hold in order to guarantee soundness of the overall workflow? In [34] some results are given using scenarios specified in terms of sequence diagrams. Another approach based on inheritance was presented in [15]. Nevertheless, the overall problem remains unsolved.
7. *How to successfully migrate instances while dynamically changing the workflow process?*
   Workflow processes may change on the fly as a result of changing laws, reengineering efforts, etc. This often leads to the situation where cases (workflow instances) have to migrate from the old workflow process to the new workflow process. As was first demonstrated in [25] this may lead to all kind of anomalies (e.g., deadlocks, livelocks, etc.). Consider a case that is moved from a process with A and B in parallel to a process with A and B in sequence. If the case is in a state after executing B but before executing A, there is no corresponding state in the sequential process. This phenomenon is known as the "dynamic change bug" [25]. In [8] it is demonstrated that these anomalies can be avoided if the old and the new process are in a subclass/superclass relation. In [16] the problem is tackled for workflow processes without loops. Despite these partial solutions, the more general problem remains unsolved.

8. *How to incorporate structural, historical, and actual data to predict the future performance of a process?*
   When it comes to performance analysis, most approaches focus on the steady-state of a process using assumptions about the arrival process (e.g., a Poisson arrival process) and the processing times (e.g., a Gamma distribution). The problem is that such an analysis requires additional modeling, does not exploit the information available, and cannot be used for short-term decisions. Business process management systems contain information about the structure of the process (e.g., a workflow process or business rules), historical data (e.g., transaction logs), and actual data (e.g., the state of each case). In principle, this information can be used to automatically generate performance models to predict throughput times and utilization rates in the near future. A workflow net augmented with predictions about the arrival of new cases, the processing times of tasks, and routing probabilities (note that such settings can be derived from historical data) can be used to generate a simulation model. If this simulation model is initialized using the current marking of the net, the near future can be analyzed. Such an approach requires a mixture of structural, historical, and actual data [40]. One of the problems is that these different kinds of data may overlap or may even be conflicting. Another problem is that most tools are designed for steady-state analysis rather than the analysis of the transient behavior. These and other problems need to be tackled in order to exploit the information in today's business process management systems.
9. *How to calculate the throughput time of concurrent workflow processes?*
   When it comes to the analysis of processes with parallelism, little has changed since the landmark paper by Baskett et al. [18] in 1975. Most of the fundamental problems remain. Queueing networks are particularly suitable for sequential processes [21] and stochastic nets [38] are typically restricted to phase-type distributions and suffer from the state-explosion problem. An alternative approach was presented in [10]. This approach allows for arbitrary discrete-time distributions provided that the workflow net can be constructed using sequential, parallel, conditional, and/or iterative routing. The problem with this approach is that it becomes intractable in case of iteration and long-tailed distributions. Approaches based on simulation are typically time-consuming and do not give exact results. The problem remains that there is no analytical technique which can cope with parallelism and non-phase-type distributions.
10. *How to allocate a fixed set of resources to tasks to optimize performance?*
    Most techniques for performance analysis are of type "What if?" and assume a given distribution of resources over tasks. Clearly, an optimal distribution of resources can be achieved by a "What if?" analysis of all possible resource allocations. Unfortunately, such an approach is intractable for large numbers of resources and tasks. An interesting alternative approach was given by Goldratt [28]. He proposes to first allocate the minimal number resources to each task. This can be determined by simply taking the product of the frequency of a task and the average processing time. The additional

resources are distributed one-by-one using the following mechanism: Determine the bottleneck and allocate one free resource to this bottleneck. Then again determine the bottleneck after adding this resource and allocate the next free resource. This procedure is repeated until all free resources have been allocated. Clearly, this procedure is much more efficient than analyzing all possible resource allocations. Unfortunately, as shown in [30], this procedure does not lead to an optimal distribution. In [30] it is shown that for various interpretations of the term bottleneck (e.g., average queue time and resource utilization) there exist counterexamples for Goldratt's algorithm. Van Hee and Reijers also propose a so-called marginal allocation strategy. This strategy is optimal for state-machine workflow nets with Poisson arrivals and negative exponential service times [30]. Unfortunately, all known strategies fail in the presence of parallelism and/or non-exponential service times.

As indicated before, we do not claim that this list of problems is of the same stature as Hilbert's problems. In fact, the list is heavily biased by personal experiences. Some problems are very concrete (e.g., the complexity of deciding soundness) while others are more general, and therefore, less concrete (e.g., how to allocate a set of resources).

7 Conclusion

In this paper, the application of Petri nets to business process management was discussed. First, business process management was put in its historical perspective. Then, the topics of process design, process analysis, and process enactment were discussed. These topics have been illustrated using Petri nets. To conclude, in the spirit of Hilbert's problems, 10 interesting problems for people working on Petri-net theory have been posed. We hope that the challenges mentioned in this paper will stimulate new and exciting research.

References

1. W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. 6, 8, 9, 10, 15
2. W. M. P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999. 9
3. W. M. P. van der Aalst. Woflan: A Petri-net-based Workflow Analyzer. *Systems Analysis - Modelling - Simulation*, 35(3):345–357, 1999. 11
4. W. M. P. van der Aalst. Loosely Coupled Interorganizational Workflows: Modeling and Analyzing Workflows Crossing Organizational Boundaries. *Information and Management*, 37(2):67–75, March 2000. 11
5. W. M. P. van der Aalst. Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, Berlin, 2000. 15

6. W. M. P. van der Aalst. *Making Work Flow: On the Design, Analysis and Enactment of Business Processes (inaugural lecture given at 30 November 2001)*. Eindhoven University of Technology, Eindhoven, The Netherlands, 2001. 1
7. W. M. P. van der Aalst and T. Basten. Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In J.M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 32–52. Springer-Verlag, Berlin, 2001. 15
8. W. M. P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, 2002. 11, 17
9. W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000. 4
10. W. M. P. van der Aalst, K.M. van Hee, and H.A. Reijers. Analysis of Discrete-time Stochastic Petri Nets. *Statistica Neerlandica*, 54(2):237–255, 2000. 18
11. W. M. P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. *Information Systems*, 25(1):43–69, 2000. 11
12. W. M. P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns Home Page. http://www.tm.tue.nl/it/research/patterns/. 16
13. W. M. P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000. 5, 8, 16
14. W. M. P. van der Aalst and S. Jablonski, editors. *Flexible Workflow Technology Driving the Networked Economy*, Special Issue of the International Journal of Computer Systems, Science, and Engineering, volume 15, number 5. CRL Publishing Ltd, 2000. 4
15. W. M. P. van der Aalst and M. Weske. The P2P approach to Interorganizational Workflows. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 140–156. Springer-Verlag, Berlin, 2001. 17
16. A. Agostini and G. De Michelis. Improving Flexibility of Workflow Management Systems. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, Berlin, 2000. 17
17. J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, Cambridge, 1990. 8
18. F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, Closed and Mixed Networks of Queues with Different Classes of Customers. *Journal of the Association of Computing Machinery*, 22(2):248–260, April 1975. 18
19. T. Basten and W. M. P. van der Aalst. Inheritance of Behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001. 10, 15
20. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, MA, USA, 1998. 8
21. J.A. Buzacott. Commonalities in Reengineerd Business Processes: Models and Issues. *Management Science*, 42(5):768–782, 1996. 18

22. J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998. 17
23. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995. 10
24. C.A. Ellis. Information Control Nets: A Mathematical Model of Office Information Flow. In *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, Colorado, 1979. ACM Press. 4
25. C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, editors, *Proceedings of the Conference on Organizational Computing Systems*, pages 10 – 21, Milpitas, California, August 1995. ACM SIGOIS, ACM Press, New York. 17
26. C.A. Ellis and G. Nutt. Workflow: The Process Spectrum. In A. Sheth, editor, *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, pages 140–145, Athens, Georgia, May 1996. 4
27. J.W. Forrester. *Industrial Dynamics*. MIT Press, Cambridge, MA, 1968. 5
28. E.M. Goldratt and J. Cox. *The Goal*. Gower, Aldershot, UK, 1984. 18
29. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987. 8
30. K.M. van Hee, H.A. Reijers, H.M.W. Verbeek, and L. Zerguini. On the Optimal Allocation of Resources in Stochastic Workflow Nets. In K. Djemame and M. Kara, editors, *Proceedings of the Seventeenth UK Performance Engineering Workshop*, pages 23–34. University of Leeds, Leeds, UK, 2001. 19
31. D. Hilbert. Mathematical problems. *Bulletin of the American Mathematical Society*, 8:437–479, 1902. 2, 15
32. A. W. Holt. Coordination Technology and Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, volume 222 of *Lecture Notes in Computer Science*, pages 278–296. Springer-Verlag, Berlin, 1985. 4
33. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996. 3, 5
34. E. Kindler, A. Martens, and W. Reisig. Inter-Operability of Workflow Applications: Local Criteria for Global Soundness. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 235–253. Springer-Verlag, Berlin, 2000. 17
35. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, Special Issue of Computer Supported Cooperative Work, 2000. 4
36. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997. 3, 4, 16
37. P. Lyman and H. Varian. How Much Information. http://www.sims.berkeley.edu/how-much-info. 13
38. M. Ajmone Marsan, G. Balbo, and G. Conte et al. *Modelling with Generalized Stochastic Petri Nets*. Wiley series in parallel computing. Wiley, New York, 1995. 18
39. C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962. 6
40. H.A. Reijers and W. M. P. van der Aalst. Short-Term Simulation: Bridging the Gap between Operational Control and Strategic Decision Making. In M.H. Hamza,

editor, *Proceedings of the IASTED International Conference on Modelling and Simulation*, pages 417–421. IASTED/Acta Press, Anaheim, USA, 1999. 18
41. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998. 10
42. W. Sadiq and M.E. Orlowska. Analyzing Process Models using Graph Reduction Techniques. *Information Systems*, 25(2):117–134, 2000. 9, 11
43. H.M.W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001. 11, 12, 15
44. T. Weijters and W. M. P. van der Aalst. Rediscovering Workflow Models from Event-Based Data. In V. Hoste and G. de Pauw, editors, *Proceedings of the 11th Dutch-Belgian Conference on Machine Learning (Benelearn 2001)*, pages 93–100, 2001. 17
45. Woflan Home Page. http://www.tm.tue.nl/it/woflan. 11
46. M.D. Zisman. *Representation, Specification and Automation of Office Procedures*. PhD thesis, University of Pennsylvania, Warton School of Business, 1977. 4

Model Validation – A Theoretical Issue?

Jörg Desel

Katholische Universität Eichstätt-Ingolstadt, Lehrstuhl für Angewandte Informatik
85071 Eichstätt, Germany
joerg.desel@ku-eichstaett.de

Abstract. The analysis and verification of a Petri net model can only yield a valuable result if the model correctly captures the considered system and if the analyzed or verified properties reflect the actual requirements. So validation of both nets and specifications of desired properties is a first class task in model-based system development. This contribution considers validation concepts based on various chapters of Petri net theory. A particular emphasis is on simulation based validation. Simulation means construction of runs, which are high-level process nets in our approach. We discuss how simulation is used for validation purposes and how the creation of runs can be performed in an efficient way.

1 Introduction: What Model Validation is About

It is often claimed that the most important features of Petri nets are their graphical representation of models, their executability and their mathematical foundation (see the discussion in [5]). The graphical representation is said to support the intuitive understanding of the model. The fact that Petri nets can be executed allows simulation of Petri nets. The precise mathematical foundation is a prerequisite for the huge variety of analysis techniques developed for Petri nets. Terms frequently used in system development processes such as *verification*, *validation*, and *evaluation* are somehow related to these features. The aim of this paper is to state these relations more explicitly, with a particular emphasis on validation. More precisely, we aim at showing how different concepts developed within Petri net theory can be used for validating Petri net models. In particular, an approach which is based on results obtained within the *VIP*[1] *project* ([3,4]) is presented (the VIP project has a particular focus on business processes and information systems but its results can easily be generalized to other application domains). This approach is based on partial order simulation and analysis of the constructed runs, formally represented by process nets. It considers place/transition nets as well as high-level Petri nets.

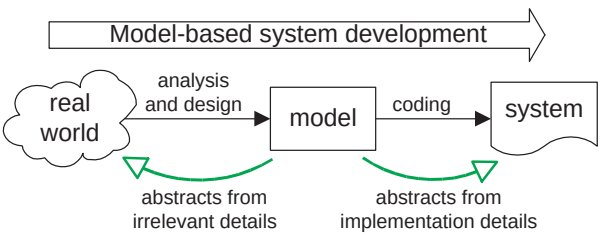
Before starting, it is useful to fix terminology. The usual definition of validation of a *system* in relation to verification and evaluation reads as follows:

[1] "Validation of Information Systems by Evaluation of Partially Ordered Runs", supported by Deutsche Forschungsgemeinschaft (DFG)

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 23–43, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Validation. Validation is the process determining that the system fulfills the purpose for which it was intended. So it should provide an answer to the question *"Did we build the right system?"*. In the negative case, validation should point out which aspects are not captured, or any other mismatch between the system and the actual requirements.
Verification. Verification is the automated or manual creation of a proof showing that the system matches the specification. A corresponding question is *"Did we build the system right?"*. In the negative case, verification should point out which part of the specification is not satisfied and possibly give hints why this is the case, for example by providing counter examples. Nowadays, *model checking* is the most prominent technique used for automated verification. *Proof techniques* can be viewed as manual verification methods.
Evaluation. Evaluation concerns the questions *"Is the system useful?"*, *"Will the system be accepted by the intended users?"*. It considers those aspects of the system within its intended environment that are not formulated or cannot be formulated in terms of formal requirements specifications. The question *"How is the performance of the system?"* might also belong to this category, if the system's performance is not a matter of specification.

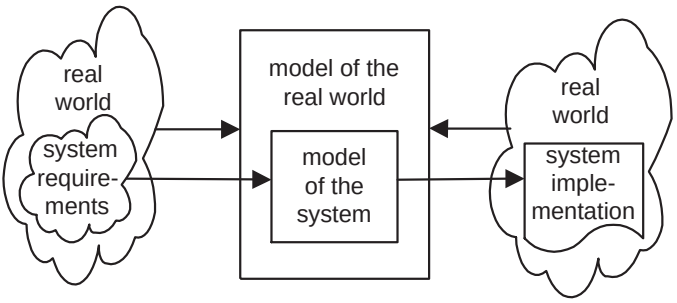
This contribution is about validation of *models*, namely Petri net models. So replacing the term "system" in the above definitions by "Petri net model" should provide the definitions we need. Models are frequently used as specifications of systems. Unfortunately, replacing "system" by "specification" in the above definitions does not make much sense. So we need a more detailed investigation of the role of models and of validation in model-based system development. The following figure presents the usual view ([9]):



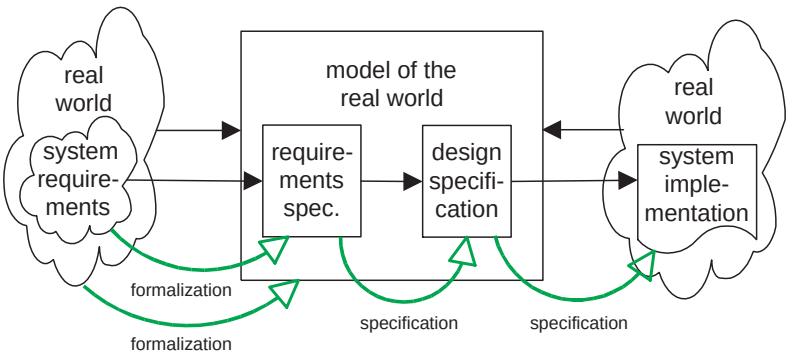
In this figure, the model is an abstract representation of both, the relevant part of the "real world" and the actual system implementation. It abstracts from irrelevant details of the considered part of the "real world", and it abstracts from implementation details of the system. Verification mainly concerns the relation between the model and the system implementation, validation concerns the relation between the model and the "real world", whereas evaluation directly relates the system and the "real world".

The above view ignores that the system to be implemented will have to function within an environment, which also belongs to the "real world". So the left hand side and the right hand side of the picture cannot be completely

separated; they are linked via the "real world". The following figure shows a more faithful representation of the situation.



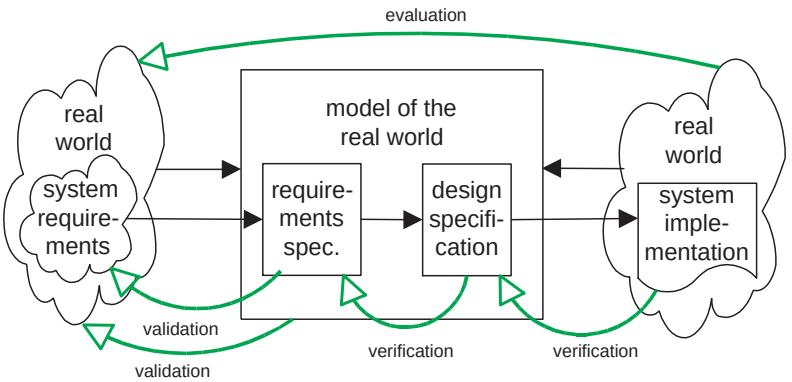
A more detailed view of the model distinguishes *requirements specifications* and *design specifications* on the model level.



The model of the real world is obtained by analysis of the domain and *formalization* of its relevant aspects. The requirements specification models the requirements and is derived by *formalization* of the requirements that exist within the "real world". The design specification can be viewed as a model of the system implementation, without considering implementation details though. It has to satisfy all properties formulated in the requirements specification. The transformation from the requirements specification to the design specification is a nontrivial task. Finally, there should be a more or less direct transformation from the design specification to the system implementation. Again, the implementation is said to be *specified* by the design specification.

Now let us consider the reverse direction. It is a matter of *verification* to check whether the design specification actually matches the requirement specification. It can also be *verified* whether the system implementation reflects the design specification. The correctness of the formalization transformations can only be checked by *validation*. So "formalization" and "validation" is a related pair of terms in the same sense as "specification" and "verification". Finally, requirements that are not captured in the model can only be checked by *evaluation* of the system implementation within the "real world".

In the following figure, the arrow annotated by "evaluation" points to the "real world" including the system requirements whereas the lower arrow annotated by "validation" addresses the real world without system requirements.



The paper is organized as follows: Section 2 presents concepts to validate Petri net models. Section 3 is devoted to formal definitions of place/transition nets and their causal semantics. Section 4 considers simulation by construction of process nets. Section 5 is about validation of requirements specifications, namely of *goals* and *facts*. Section 6 generalizes the previous results to high-level Petri nets. It provides a definition of abstract process nets that allow an efficient validation of high-level Petri nets.

2 Validation of Petri Net Models

Let us now turn to Petri nets. A considerable part of the huge Petri net literature considers the problem

> *"Given a Petri net model and some properties formulated in an appropriate way, verify or prove that the model satisfies these properties."*

The properties can be "standard" properties of Petri nets such as liveness or boundedness, or they can be specific for the considered model and formulated in a general-purpose specification language such as temporal logics. So this question mainly addresses the specification/verification relation between the requirements specification and the design specification.

How can Petri net theory help in validating Petri net models? Or, stated the other way, how can Petri net theory help to avoid errors in formalizing concepts in terms of Petri nets? We shall call the relevant part of the "real world" together with the requirements *system* in the sequel[2]. So our question is:

> *"Given a Petri net model and some properties, how can we validate that both reflect the actual system?"*

[2] There is some confusion with the usage of "system" in the first figure of the introduction; however, in both settings "system" is the usual term.

Graphics. The first and most obvious answer is that the graphical visualization of a Petri net, of local vicinities of its elements, of its components and of the communication primitives between components reflect the structure of the system. This is an important advantage of Petri nets compared to competing modeling languages. However, the graphical representation only helps to avoid modeling errors, but cannot prevent modeling errors. For the formalization and graphical representation of requirements there exist suggestions to include additional elements in a Petri net representing single requirements (e.g., *fact-transitions*, defined in [11]). These elements have limited expressive power but, by their integration in the Petri net model, allow a very clear and intuitive specification of requirements, whereas general temporal logic formulas tend to be quite complicated and error prone.

Structural Analysis. Structural analysis of Petri net models helps finding modeling errors, i.e., errors where the model does not reflect the wrong system but no reasonable system at all. Place invariant analysis ([2]) is a typical example. If some place is not covered by a semi-positive place invariant then it is possible that this place is erroneously unbounded. If a variable of a distributed algorithm is represented by some set of places, then these places should constitute a place invariant, representing the fact that there is always exactly one token on these places ([15]) (this is not a system requirement). Another example for a structural analysis technique is given by mismatching splits and joins. It often is unreasonable that two paths split at a transition (AND-split) but join at a place (OR-join) or vice versa. The identification of such a situation can be a valuable help in finding modeling errors ([16]).

Behavioral Analysis. There exist well established notions within Petri net theory for standard behavioral properties, e.g. liveness, deadlock-freedom, boundedness, safety and reversibility (see [7] for definitions concerning place/transition nets). Often it is obvious that a system model should enjoy some or all of these properties although they are not part of the specification. So, by behavioral analysis, ill-designed Petri net models can be identified. However, in many cases behavioral analysis techniques are too complex to be applicable to Petri net models of industrial-sized systems.

Simulation. By simulation we understand the generation of runs of the model. For a valid Petri net model, each run should represent a corresponding run of the system, and for each system run there should exist a corresponding run of the model. Validation by simulation means generating and inspecting runs of the model with respect to the desired runs of the modeled system. Since neither the system nor its runs are given formally, only domain experts can do this comparison. So this task requires a good and easy understanding of the generated runs of the model. Usually, the user is supported by a graphical representations of runs: The Petri net is represented graphically and runs are depicted by subsequent occurrences of transitions of the net.

The approach presented in the remainder of this contribution is on validation by simulation. We suggest to construct and visualize causal runs given by partially ordered *process nets* instead of sequential runs given by occurrence sequences. Before presenting this approach and its advantages, we provide definitions and elementary results in the following section.

3 Causal Semantics of Place/Transition Nets

Each system has a dynamic behavior, given by its set of *runs*. In a run, *actions* of the system can occur. We will distinguish actions from *action occurrences* and call the latter *events*. In general, an action can occur more than once in a single run. Therefore, several events of a run might refer to the same action.

There are basically two different techniques to describe the behavior of a Petri net model: A single run can either be represented by a sequence of action names (a sequence of sets of action names, respectively) representing subsequent events (sets of events, respectively) or by a causally ordered set of events. The first technique is formally described by *occurrence sequences* (*step sequences*, respectively). It constitutes the *sequential semantics* of a Petri net. The second technique employs *process nets* representing *causal runs*. It constitutes the *causal semantics* of a Petri net.

The main advantage of sequential semantics is formal simplicity. Sequential semantics generalizes well-known concepts of sequential systems; every occurrence sequence can be viewed as a sequence of global system states and transformations leading from a state to a successor state. One of the main advantages of causal semantics is its explicit representation of causal dependency, represented by paths of directed arcs in process nets. Consequently, concurrent events are events that are not connected by a path in a process net.

Causal semantics of Petri nets has been studied in Petri net theory since a long time, starting with the work of Carl Adam Petri in the seventies (see [1] for an overview). More recently, variants of causal semantics are used for efficient verification algorithms ([10]). Application projects employing Petri nets, however, mostly restrict to sequential semantics, and so do most Petri net tools.

In sequential semantics, a run is represented by a sequence of events such that causal dependencies are respected; if an event causally depends on another event, then these events will not appear in the reverse order in an occurrence sequence. A causal run also consists of a set of events, representing action occurrences of the system. An action can only occur in certain system states, i.e. its pre-conditions have to be satisfied. The occurrence of the action leads to a new system state where some post-conditions of the action start to hold. An event is therefore causally dependent on certain pre-conditions and might lead to new conditions that are causal prerequisites for other events. The time and the duration of an event has no immediate influence on the system's behavior, as long as such dependencies are not explicitly modeled as actions of clocks. Combining events with their pre- and post-conditions yields a causal run, formally represented by a *process net*. Since pre- and post-conditions of events are explicitly modeled

in a process net, the immediate causal dependency is represented by the arcs of a process net. The transitive closure of this relation defines a partial order that we will call *causal order*; two events are causally ordered if and only if they are connected by a chain of directed arcs. Otherwise, they are not ordered but occur *concurrently*.

Using acyclic graphs to define partially ordered runs is common for many computation models. The specific property of process nets is that each process net is formally a Petri net and that there is a close connection between a process net representing a run and the Petri net modeling the system; the events of a process net are annotated by respective names of actions of the system. More precisely, mappings from the net elements of the process net to the net elements of the Petri net representing the system formalize the relations between events of a process net and transitions of a system net and between conditions of a process net and places of a system net.

We roughly follow the standard definitions and notations of place/transition nets and process nets ([12,7]). However, in contrast to the usual notion, we equip process nets with initial states, represented by markings of conditions. We also do not consider extensions such as inhibitor arcs (requiring that a place contains no token to enable a transition), nets with preemptive behavior (where all tokens of a place are removed by a transition occurrence) or timed Petri nets.

A place/transition net $(P, T, pre, post, M_0)$ is given by

- a finite set P (places),
- a finite set T satisfying $P \cap T = \emptyset$ (transitions),
- two mappings $pre, post: T \times P \to \{0, 1, 2, \ldots\}$,
  the derived flow relation F is given by

$$F = \{(p,t) \in P \times T \mid pre(t,p) > 0\} \cup \{(t,p) \in T \times P \mid post(t,p) > 0\} \ ,$$

- an initial marking $M_0: P \to \{0, 1, 2, ...\}$ (represented by tokens in the places).

For a net element x in $P \cup T$, ${}^\bullet x$ (the pre-set of x) denotes the set of elements y satisfying $(y, x) \in F$ and $x^\bullet$ (the post-set of x) denotes the set of elements y satisfying $(x, y) \in F$. We restrict our considerations to place/transition nets without transitions t satisfying ${}^\bullet t = \emptyset$ or $t^\bullet = \emptyset$.

Given an arbitrary marking $M: P \to I\!N$, a transition t is enabled if each place p satisfies $pre(t,p) \leq M(p)$. The occurrence of t leads to a new marking M', defined for each place p by

$$M'(p) = M(p) - pre(t,p) + post(t,p) \ .$$

We denote the occurrence of t at the marking M by $M \xrightarrow{t} M'$.

The causal behavior of a place/transition net $(P, T, pre, post, M_0)$ is defined by its set of process nets, representing causal runs. For the formal definition of a process net, we employ again place/transition nets: Each process net of the place/transition net $(P, T, pre, post, M_0)$ is given by a place/transition net

$(C, E, pre', post', S_0)$, together with mappings $\alpha: C \to P$ and $\beta: E \to T$, satisfying the conditions given below. The places of a process net are called *conditions*, the transitions *events* and the markings *states*. To avoid confusion with process nets, the place/transition net model $(P, T, pre, post, M_0)$ of the system will be called *system net* in the sequel.

- Every condition c in C satisfies $|{}^\bullet c| \leq 1$ and $|c^\bullet| \leq 1$,
- for each event e in E and each condition c in C we have $pre'(e, c) \leq 1$ and $post'(e, c) \leq 1$ (hence the flow relation, which we denote by K here, completely determines pre' and $post'$),
- the transitive closure of the flow relation K is irreflexive, i.e., it is a partial order over $C \cup E$,
- for each event e in E and for each place p in P we have

$$|\{c \in {}^\bullet e \mid \alpha(c) = p\}| = pre(\beta(e), p),$$

$$|\{c \in e^\bullet \mid \alpha(c) = p\}| = post(\beta(e), p).$$

- $S_0(c) = 1$ for each condition c in C satisfying ${}^\bullet c = \emptyset$ and $S_0(c) = 0$ for any other condition c in C,
- $\alpha(S_0) = M_0$, where α is generalized to states S by

$$\alpha: (C \to \mathbb{N}) \to (P \to \mathbb{N}), \quad \alpha(S)(p) = \sum_{\alpha(c)=p} S(c).$$

A condition c in C represents the appearance of a token on the place $\alpha(c)$. An event e in E represents the occurrence of the transition $\beta(e)$. In a run, each token is produced by at most one transition occurrence, and it is consumed by at most one transition occurrence. Hence, conditions of process nets are not branched. The transitive closure of K defines the *causal relation* on events and conditions. Since no two elements can be mutually causally dependent, the causal relation is a partial order. In other words, the flow relation has no cycles. Since events represent transition occurrences, the pre- and post-sets of these transitions are respected. The initial state of the process net is the characteristic mapping of the set of conditions that are minimal with respect to the causal order, i.e., these conditions carry one token each, and all other conditions are initially unmarked. Note that all minimal elements are conditions because, by our general assumption, every event has at least one pre-condition. Finally, the initial state of the process net corresponds to the initial marking of the system net, i.e., each initial token of the system net is represented by a (marked) minimal condition of the process net.

As mentioned before, a process net represents a single causal run of a system net. We equip a process net with an initial state S_0 so that the token game can be played for process nets as well. As will be stated in the following well-known lemma, the sequences of event occurrences of a process net closely correspond to transition sequences of the system net.

Remark 1. In a process net, every event can occur exactly once. The order of these event occurrences respects the causal relation. Every reachable state is *1-safe*, i.e., no condition ever carries more than one token. Moreover, in every reachable state, no two marked conditions are ordered by the causal relation.

Lemma 1 ([1]). *Let (P, T, F, M_0) be a place/transition net. If (C, E, K, S_0) together with mappings $\alpha: C \to P$ and $\beta: E \to T$ is a process net and*

$$S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} S_n$$

is a sequence of event occurrences, then

$$M_0 \xrightarrow{\beta(e_1)} \alpha(S_1) \xrightarrow{\beta(e_2)} \cdots \xrightarrow{\beta(e_n)} \alpha(S_n)$$

is a sequence of transition occurrences of (P, T, F, M_0).

Conversely, for each sequence

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$$

of transition occurrences of the place/transition net (P, T, F, M_0), there is a process net (C, E, K, S_0) with mappings $\alpha: C \to P$ and $\beta: E \to T$ and a sequence of event occurrences

$$S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} S_n$$

such that, for $1 \leq i \leq n$, $\alpha(S_i) = M_i$ and $\beta(e_i) = t_i$.

The first part of the lemma follows immediately from the definition of process nets because transition vicinities are respected by the mappings α and β. For proving the converse direction, we can successively construct a suitable process net by adding events with pre- and post-conditions according to the occurring transitions (see [14]). In general this construction is not unique: Consider a place p carrying n tokens at a marking and the occurrence of a transition t satisfying $pre(t, p) = m < n$. Then, the process net constructed so far has n conditions with empty post-set mapped to p. Only m of these conditions will be linked to the new event e wich is mapped to t. However, the construction is unique if no reachable marking assigns more than one token to a place (i.e., if the place/transition net is *1-safe*).

Lemma 1 states that process nets respect the sequential behavior; no information about possible occurrence sequences is gained or lost when we consider process nets. Moreover, it states that reachable markings of the system net closely correspond to reachable states of its process nets, i.e., for every marking reachable in the system net there is (at least) one corresponding reachable marking of a process net. Conversely, the image of a reachable marking of a process net is reachable in the system net. The following lemma, which is also well-known, gives another characterization of reachable states of process nets.

Lemma 2 ([1]). *Let (C, E, K, S_0) be a process net of a place/ transition net. A state S is reachable from S_0 (by a sequence of event occurrences) if and only if S is the characteristic mapping of a maximal set of pairwise unordered conditions.*

Maximal sets of conditions that are mutually not ordered are called *cuts*. By Lemma 1 and Lemma 2, a marking M of a system net is reachable if and only if there exists a process net with a cut that is mapped to M (more precisely, the state given by the characteristic mapping of the cut is mapped to M).

4 Simulation by Generation of Process Nets

Traditionally, simulation of a net means construction and visualization of sequences of transition occurrences. In our approach, not event sequences but process nets are constructed. We argue that we gain two major advantages, namely expressiveness and efficiency.

Expressiveness. Every sequence of events, i.e. transition occurrences, defines a total order on these events. A transition can either occur after another transition because there is a causal dependency between these occurrences or the order is just an arbitrarily chosen order between concurrent transition occurrences. Hence, an occurrence sequence gives little information on the causal structure of the system run. Interesting aspects of system behavior such as the flow of control, the flow of goods, possible parallel behavior etc. are directly represented in process nets, but they are hidden in sequences of events.

Efficiency. Simulation of a system model means construction of a set of (different) runs. In general, each causal run corresponds to a nonempty set of occurrence sequences. This correspondence is formally established by Lemma 1; taking the sequence of labels of events in occurrence sequences of process nets yields all occurrence sequences of the system net. The number of event occurrence sequences of a single process net grows dramatically when a system exhibits more concurrency. Each of these occurrence sequences represents the very same causal system run. Hence, the simulation of more than one of these sequences yields no additional information on the causal behavior of the system. The gain of efficiency is most evident when all runs of a system can be simulated, i.e. when there is only a finite number of finite runs. In the case of arbitrary large runs, a set of process nets allows to represent a larger significant part of the behavior than a comparable large set of occurrence sequences.

How does a simulation algorithm constructing process nets work? The core idea is to start with a set of marked conditions that correspond to the initial marking and then subsequently add suitable events, add arcs to their preconditions, add post-conditions, and add arcs from the new event to the post-conditions (see [14]). In other words, we consider an occurrence sequence of the system net and construct a process net such that this occurrence sequence is a sequence of labels of the occurrence sequence of the process net. The existence of a suitable process net is guaranteed by Lemma 1.

Alternatives. Transition occurrences can exclude each other. More precisely, at some reachable marking, the occurrence of a transition might disable another transition and vice versa. During the construction of a process net, this means that there is an alternative between two events that can be added to a process net constructed so far, and both events have a common pre-condition. For the construction of different process nets, possible alternative branches in process nets constructed so far have to be found. To this end, we construct a *branching process* from a set of process nets. Every two different process nets share a unique maximal common prefix (including the conditions representing the initial marking). More precisely, up to some state they are isomorphic, but they have different continuations. A branching process can be generated from a set of process nets by identification of their maximal common prefixes. Since different continuations split at places, branching processes are allowed to have forward branching places. The initial marking of a branching process is given by the common initial marking of its generating process nets. As the elements of its process nets, elements of a branching process are mapped to elements of the system net.

A new alternative is obtained by finding a reachable state S of the actual branching process such that some event e of the branching process net is enabled at S, the corresponding marking of the system net enables a transition t such that $\beta(e) \neq t$ and $\beta(e)$ and t share at least one input condition. We then create a new process net by copying the part of the branching process up to the places marked at S (which is a process net) and adding a new event e' representing the occurrence of t.

The branching processes defined here are similar to branching processes used in [10]. The main difference is that branching processes in [10] are constructed in a breadth first style while our branching processes are constructed depth first.

Termination. In general, a process net can be continued arbitrarily if it does not lead to a state representing a deadlocked marking. But simulation must eventually terminate each process net generation. For validation purposes, it does not make much sense to stop the construction of a process net at an arbitrary state. Instead, we distinguish *internal* and *external* Petri net transitions (see [15]). An internal transition represents an activity that *must* be executed; the transition eventually has to occur, unless it is disabled by another transition occurrence. External transitions represent interface activities that need external input. They always *can* occur but *may* remain enabled forever without occurring.

Roughly speaking, in our simulation approach we terminate a process net at a state where the corresponding marking of the system net enables no internal transition. So we start with the occurrence of internal and external transitions. Eventually we stop the any occurrences of external transitions and continue with internal transitions only. For systems that require repeated interaction with the environment, the simulation of the model will eventually terminate.

This termination criterion does not work if there are process nets which do not always again involve external transitions. In this case it is possible that, after

some prefix, arbitrarily many events representing internal transitions occur, and prohibiting the occurrence of external transitions has thus no effect.

If a system net has only finitely many different reachable markings then each arbitrarily growing process net has subsequent reachable states representing the same marking. The events between the states constitute a cyclic behavior, and arbitrary repetition of this cycle will not yield any new information. The following termination criterion employs this fact.

We associate to each event e of a process net the unique earliest reachable state S_e after the occurrence of e. The state S_e is reached by the occurrence of all events causally before e (these events can occur in any total order respecting the partial order given by the process net) and the occurrence of the event e itself. The state S_e is mapped to a marking M_e of the system net. If, for some event e', there is a path leading from e' to e (i.e., e' causally precedes e), and the associated markings coincide (i.e., $M_e = M_{e'}$), then every continuation of the process net after reaching the state S_e could have happened after $S_{e'}$ as well. As a consequence, it is not necessary to generate the event e. The event e is called a *cut-off event*. The concept of cut-off events referring to other events in the same process net was inspired by cut-off transitions of branching processes ([10]).

After finding a cut-off event, the process net generation algorithm does not necessarily stop because possibly still non-cut-off events can be added. Next we show that – at least for *bounded* system nets which have only a finite set of different markings – the cut-off technique always eventually leads to termination.

Theorem 1. *For each process net without cut-off events of a bounded system net there is an upper bound for the number of events.*

Proof. Consider a sequence of events $e_1, \ldots, e_k$ of a process net such that, for $1 \leq i \leq k-1$, the post-set of e_i and the pre-set of e_{i+1} share a common condition. Then all events belongs to a directed path. The respective states S_{e_i} are mapped to markings M_{e_i}. Since the process net has no cut-off event, all these markings M_{e_i} are different. Since the set of reachable markings is finite, the number of different reachable markings of the system net is an upper bound for k. So process nets cannot have arbitrarily long paths. Since a process net starts with a finite set of conditions and since each element has only finitely many immediate successors, the number of elements of the process net is bounded, too. □

For simplicity, we ignore external transitions, i.e., we consider the process construction when they are no more allowed to occur. Then the previous theorem can only be applied if the number of markings reachable without firing external events is always finite, which is a reasonable requirement for most systems. What happens otherwise? The cut-off technique for constructing process nets is useful in this case, too. Using the notation of the previous proof, an arbitrarily growing process net possesses an arbitrarily growing sequence $e_1, e_2, \ldots$ of ordered events and thus an arbitrarily growing sequence of markings M_{e_i}. All these markings are different – otherwise a cut-off event would have been identified. It is well known that such a sequence eventually contains markings M_{e_k} and M_{e_l}, $k < l$,

such that $M_{e_k}(p) \leq M_{e_l}(p)$ for each place p and $M_{e_k}(p) < M_{e_l}(p)$ for some place p (see e.g. the proof of the finiteness of the *coverability graph* in [7]). In this case, the events leading from S_{e_k} to S_{e_l} can occur at S_{e_l} again, and this construction can be arbitrarily iterated. Since the total number of tokens is increased by the occurrence of these events, it grows arbitrarily. Therefore, the number of reachable markings is infinite in this case. So we either find cut-off events leading to termination, or we identify the system net as unbounded.

5 Validating Requirements

A Petri net model of a system can be equipped with formal requirements that the model should satisfy. For simulation-based validation of these requirements we construct process nets and check whether these runs contradict the requirements specification. In the negative case, either the formalization of requirements or the model is not correct, which can be investigated further by the user.

An iterative construction of the model (the design specification) from requirement specifications uses models and requirements that are not (yet) "implemented" in the model. Again, process nets are constructed, and it is checked whether a certain requirement specification holds for this run. The user receives a separation of positive and negative runs which helps to find out whether the formal requirements specification matches the actual system requirements.

Therefore, we provide means to formulate required properties and to identify violations in simulated runs. In our approach, the specification of the properties is done in the (graphical representation of the) system model (see [3]), adopting the well-known fact transitions ([11]) and introducing analogous graphical representations for other properties. In the present paper we only consider two classes of properties, namely *goals* and *facts*.

Goals. A goal requires that a certain situation will eventually occur. More precisely, each process net satisfying some fairness condition to be defined later should contain a condition that is mapped to a certain place. Clearly, checking whether a given process net contains such a condition is a trivial task.

If the simulation of process nets can be stopped at arbitrary states then process nets provide little information about goals; if a goal is not reached in a process net then this required property either does not hold true, or it holds true and a state marking a corresponding condition is reached in any continuation of the stopped process net. So, for analysis of goals, we have to reconsider termination conditions for process net construction. As mentioned before, we allow termination at states corresponding to markings that enable only external transitions. So reaching a goal should not depend on external transitions, which is very reasonable. Instead, we ignore external transitions in the sequel.

If the system net does not enjoy the property formulated by a goal then there either is a finite process net that terminates at external transitions or an arbitrarily growing process net which has no such condition. The termination condition is only concerned with the second case. So we need a formal treatment

of arbitrarily growing process nets. We have to make a fairness assumption with respect to internal transitions, since they eventually have to occur.

A (finite or infinite) sequence of process nets $P_1, P_2, P_3, \ldots$, where P_{i+1} is obtained from P_i by the addition of an event and its post-set (for $i \geq 1$), is called *fair process sequence* (*fair process sequence without cut-off events, respectively*) if it satisfies the following property: If some event e can be added to a process net P_i using a set of conditions C as its pre-set then there exists a process P_j with an event e' which has at least one condition of C in its pre-set. In other words, each enabled transition will either eventually occur or one of its input tokens is used by another transition. For fair process sequences without cut-off events we restrict this requirement to events e which are not cut-off events.

The following theorem states that stopping the construction of a process net at cut-off events does not hide subsequent conditions mapped to the goal place.

Theorem 2. *Consider a bounded system net with a place p. If, for some fair process sequence without cut-off events, none of its process nets has a condition mapped to p then there exists a (general) fair process sequence such that none of its process nets has a condition mapped to p, i.e., then p is not a valid goal.*

Proof. Assume a fair process sequence without cut-off events such that none of its processes has a condition mapped to p. We construct a (general) fair process sequence satisfying the same condition. Consider a cut-off event e that can be added to a process net P_i of the sequence and assume that no subsequent process net uses the conditions enabling e. By the cut-off property, the state S_e has a corresponding earlier state $S_{e'}$ mapped to the same marking of the system net. Abusing notation, we call the set of conditions marked at a state S also S. We can cut the process net at the state S_e and add the part of the process net between $S_{e'}$ and S_e as often as we like. This results not necessarily in a fair sequence of process nets because there might exist conditions which are not touched by the part between $S_{e'}$ and S_e and which can be used as the pre-set of another event. These conditions belong to the set $S_{e'} \cap S_e$.

So, instead of adding the part between $S_{e'}$ and S_e, we first freeze the conditions in $S_e \setminus S_{e'}$, i.e., we do not add any event to their post-sets. Then we consider a new sequence of process nets where – after P_i – only events are added which are not causally dependent on conditions in $S_e \setminus S_{e'}$. These elements are independent from the process part between $S_{e'}$ and S_e. Now we proceed recursively; if we find another cut-off event that can be added to a process net later in this sequence we again freeze the conditions touched by the repetitive part, and so on. This procedure will eventually terminate because at each step some conditions are frozen, all frozen conditions are pair-wise unordered, but an arbitrarily growing number of pair-wise unordered conditions implies a reachable state with an arbitrarily growing number of tokens by Lemma 2, which then implies an arbitrarily growing number of tokens of the system net – contradicting its boundedness.

Starting with the constructed process sequence, we add events of the intervals between states of cut-off events and their respective earlier reference events in

a cyclic manner. In this way, for each considered cut-off event e, the process part between $S_{e'}$ (e' being its reference event) and S_e is added again and again. By construction, this yields a (general) fair sequence of process nets. Moreover, since we only add events corresponding to transitions that have already occurred before, for each added condition there is a prior condition mapped to the same place. In particular, since none of the conditions of the initial process sequence was mapped to the place p, the same holds for process nets of this generated sequence. □

Facts. A fact is a property that holds for all reachable markings of a system model. Facts can be checked in runs by analyzing all reached states. This is simple for sequential runs because the markings reached during an occurrence sequence are explicitly represented. We show how certain facts can be checked more efficiently using causal runs.

Here, we only consider properties of markings requiring that not all places of a given set of places are marked simultaneously. A typical example is *mutual exclusion* of critical sections of parallel programs: Two places representing the critical sections should never be marked together. For identifying the violation of a fact in a process net it suffices to find conditions labeled by the respective places that are pair-wise not ordered. Every set of unordered conditions is a subset of a cut. Every cut corresponds to a reachable state of the process net by Lemma 2. Every state of the process net corresponds to a reachable marking of the system net by Lemma 1. Therefore, the unordered conditions prove that there is a reachable marking of the system net that marks all places, and the considered process net can be taken as a counter example.

Termination at cut-off events does not prevent detection of a fact violation, as is shown in the following theorem.

Theorem 3. *Consider a bounded system net with a reachable marking M. Some process net without cut-off event has a reachable state mapped to M.*

Proof. Since M is reachable there exists a process net with a reachable state mapped to M. Now choose a minimal (w.r.t. the number of elements) process net with this property. Clearly, the state S of this process net which is mapped to M marks exactly those conditions with empty post-set. Assume that this process net has a cut-off event e. Then there is a prior event e' such that S_e and $S_{e'}$ are mapped to the same marking. So we can cut away the part between these states, i.e., compose the part from the initial state to $S_{e'}$ and the part from S_e to S in the obvious way. The result is a smaller process net leading to the state S – contradicting the minimality of the chosen process net. □

6 High-Level Petri Nets

High-level Petri nets allow to use individual tokens instead of indistinguishable tokens of place/transition nets ([13,15]). They allow a much more compact representation of systems. In industrial applications, high-level Petri nets have

great advantages compared to place/transition nets because they combine the graphical representation of Petri nets with the possibility to capture data. In this section, we show how our approach works for high-level Petri nets and how the compact representation can be employed for an efficient validation using appropriate abstract runs. First, we need some definitions.

Given a set A, a *multiset* over A is a mapping from A to $\mathbb{N}$. Multisets generalize sets but a multiset can contain several identical copies of an element. As an example, a marking of a place/transition net can be viewed as a multiset over its set of places where the number of tokens on a place defines its multiplicity. We call a multiset over A finite if only finitely many elements of A are not mapped to 0. The set of finite multisets over A is denoted by $\mathcal{M}(A)$. The sum (and difference, if applicable) of two multisets is defined element-wise for each element of A.

A *high-level Petri net* is given by

- sets P and T, as defined as in the definition of a place/transition net,
- a set A_p of *individual tokens* for each place p in P (the *domain* of p), the union of all sets A_p will be denoted by A,
- a set of modes μ_t for each transition t in T, the union of all modes μ_t will be denoted by μ,
- for each pair (t,p) in $T \times P$, an input-mapping $i_{(t,p)}\colon \mu_t \to \mathcal{M}(A_p)$ specifying the tokens on the place p necessary for the occurrence of t in mode $m \in \mu_t$,
- for each pair (t,p) in $T \times P$, an output-mapping $o_{(t,p)}\colon \mu_t \to \mathcal{M}(A_p)$.
- an *initial marking* M_0, where an arbitrary *marking* $M\colon P \to \mathcal{M}(A)$ assigns to each place p in P a finite multiset $M(p) \in \mathcal{M}(A_p)$,

When a transition t is enabled by the input-mapping and occurs in mode $m \in \mu_t$ at a marking M, then a successor marking M' will be reached, defined for each place p by $M'(p) = M(p) - i_{(t,p)}(m) + o_{(t,p)}(m)$.

As for place/transition nets, the flow relation F can be derived from the input- and output mappings:

$$\begin{aligned} F = \{&(p,t) \in P \times T \mid \exists m \in \mu_t, a \in A_p\colon i_{(t,p)}(m)(a) > 0\} \\ \cup\ \{&(t,p) \in T \times P \mid \exists m \in \mu_t, a \in A_p\colon o_{(t,p)}(m)(a) > 0\} \end{aligned}$$

Process nets of high-level Petri nets are defined very similar as process nets of place/transition nets. Each process net of the high-level Petri net

$$(P, T, (A_p), (\mu_t), (i_{(t,p)}), (o_{(t,p)}), M_0)$$

is given by a place/transition net $(C, E, pre', post', S_0)$, together with mappings $\alpha\colon C \to (P \times A)$ and $\beta\colon E \to (T \times \mu)$, satisfying the conditions given below.

- Every condition c in C satisfies $|{}^\bullet c| \leq 1$ and $|c^\bullet| \leq 1$.
- For each event e in E and for each condition c in C we have $pre'(e,c) \leq 1$ and $post'(e,c) \leq 1$.
- The transitive closure of the flow relation is irreflexive, i.e., it is a partial order over $C \cup E$.

– For each event e in E, for each place p in P and for each $a \in A_p$ we have

$$|\{c \in {}^\bullet e \mid \alpha(c) = (p,a)\}| = i_{(t,p)}(m)(a),$$

$$|\{c \in e^\bullet \mid \alpha(c) = (p,a)\}| = o_{(t,p)}(m)(a),$$

where $\beta(e) = (t,m)$,
– $S_0(c) = 1$ for each condition c in C satisfying ${}^\bullet c = \emptyset$ and $S_0(c) = 0$ for any other condition c in C.
– For each place p in P and for each a in A we have:

$$\sum_{\alpha(c)=(p,a)} S_0(c) = M_0(p)(a) \ .$$

Using this definition, the compact representation given by high-level Petri nets is not employed by process nets. If the sets of domains and modes are finite, a high-level Petri net $(P, T, (A_p), (\mu_t), (i_{(t,p)}), (o_{(t,p)}), M_0)$ has a corresponding place/transition net $(P', T', pre, post, M_0')$, given by

– $P' = \{(p,a) \in P \times A \mid a \in A_p\}$,
– $T' = \{(t,m) \in T \times \mu \mid m \in \mu_t\}$,
– $pre((t,m),(p,a)) = i_{(t,p)}(m)(a)$ for each $((t,m),(p,a)) \in T' \times P'$,
– $post((t,m),(p,a)) = o_{(t,p)}(m)(a)$ for each $((t,m),(p,a)) \in T' \times P'$,
– $M_0'(p,a) = M_0(p)(a)$ for each $(p,a) \in P'$.

This place/transition net is called *flattening* of the high-level Petri net in [8].

Remark 2. The process nets of a high-level Petri net and the process nets of its flattening coincide (notice that we constructed the respective mappings α and β in such a way that α maps conditions to pairs of places and domain elements in both cases and that β maps events to pairs of transitions and modes in both cases. However, the co-domains of the mappings are not exactly the same.)

Process nets of high-level Petri nets distinguish every single token from a domain A, because they are also process nets of the flattening. This leads to many very similar process nets, which differ only with respect to the token identities. In particular, process nets won't be very different when the token identity has no influence on the path a token takes through a net. In general, a token can be not just an element of a set (as we defined it for simplicity sake) but has its own structure. Consider for example a high-level Petri net which models the flow of documents. These documents can be viewed as letters; the address or any other routing information is visible whereas the content of the letter has no influence on the routing activities. In a Petri net model, the letter could be modeled by a pair (a_1, a_2) where a_1 is the "visible" information relevant for the enabling of transitions whereas a_2 stands for the "invisible" part of the document, which is just moved through the net but has no influence on the processes. Obviously, simulation is more efficient if not all processes incorporating different tokens with identical "visible" parts are distinguished. This motivates to subsume different process nets which differ only with respect to their tokens.

Before defining a suitable equivalence on process nets we define isomorphism of place/transition nets in the obvious way:

$$(P, T, pre, post, M_0) \sim (P', T', pre', post', M_0')$$

if there are bijections $\varphi_P: P \to P'$ and $\varphi_T: T \to T'$ satisfying $pre(t,p) = pre'(\varphi_T(t), \varphi_P(p))$ and $post(t,p) = post'(\varphi_T(t), \varphi_P(p))$ for each pair (t,p) in $T \times P$ and $M_0(p) = M_0'(\varphi_P(p))$ for each p in P.

We call two process nets $(C, E, pre, post, S_0)$ and $(C', E', pre', post', S_0')$ of a high-level Petri net *weakly isomorphic* if they are isomorphic as place/transition nets (with φ_C being the bijection for conditions) and if there is a bijection $\varphi_A: A \to A$ (where A is the domain of the high-level Petri net) such that the respective mappings $\alpha: C \to (P \times A)$ and $\alpha': C' \to (P \times A)$ satisfy, for each condition c in C: $\alpha(c) = (p,a)$ iff $\alpha'(\varphi_C)(c) = (p, \varphi_A(a))$.

The idea of an abstract process net of a high-level Petri net is to comprise all weakly isomorphic process nets. Instead of using the bijection $\varphi_A: A \to A$ we will employ an equivalence relation on the domain A; if two process nets are weakly isomorphic (w.r.t. a bijection $\varphi_A: A \to A$) then each element a in A can be identified with $\varphi_A(a)$. So we define an equivalence relation $\sim \subseteq A \times A$. We denote by $[a]_\sim$ the equivalence class containing an element a and by $A/\sim$ the set of all equivalence classes.

An *Abstract Process Net* of a high-level Petri net $(P, T, (A_p), (\mu_t), (i_{(t,p)}), (o_{(t,p)}), M_0)$ w.r.t. an equivalence relation $\sim \subseteq A \times A$ is given by a place/transition net $(C, E, pre', post', S_0)$, together with mappings $\alpha: C \to (P \times A/\sim)$ and $\beta: E \to (T \times \mu)$, satisfying the same conditions as a process net of a high level Petri net except those concerning the places of the net (the fourth and the sixth item), which are replaced by the following:

- For each event e in E, for each place p in P and for each $[a] \in A_p/\sim$ we have

$$|\{c \in {}^\bullet e \mid \alpha(c) = (p,[a])\}| = \sum_{a' \in [a]} i_{(t,p)}(m)(a'),$$

$$|\{c \in e^\bullet \mid \alpha(c) = (p,[a])\}| = \sum_{a' \in [a]} o_{(t,p)}(m)(a'),$$

  where $\beta(e) = (t,m)$.
- For each place p and for each $[a] \in A/\sim$:

$$\sum_{\alpha(c)=(p,[a])} S_0(c) = \sum_{a' \in [a]} M_0(p)(a') \; .$$

Abstract process nets can also be viewed as process nets of accordingly abstracted high-level nets. They have some relation to high-level process nets, introduced in [8]. Given an arbitrary high-level Petri net and an equivalence relation $\sim \subseteq A \times A$, we can define an *abstract high-level Petri net* as follows:

- The sets of places and of transitions are as in the original net,

- The new domain A'_p of a place p is the set of equivalence classes containing at least one element of the old domain A_p,
- The modes of transitions are as in the original net, but the occurrence of a transition in different modes might have identical effects in the abstract net,
- Each input-mapping $i_{(t,p)}: \mu_t \to \mathcal{M}(A_p)$ is replaced by the input mapping $i'_{(t,p)}: \mu_t \to \mathcal{M}(A_p/\sim)$, defined by

$$i'_{(t,p)}(m)([a]) = \sum_{a' \in [a]} i_{(t,p)}(m)(a')$$

  for each $[a]$ in $A_p/\sim$, and similarly for the output mappings.
- The new initial marking $M'_0: P \to \mathcal{M}(A/\sim)$ is obtained from the old initial marking M_0 by

$$M'_0(p)([a]) = \sum_{a \in [a]} M_0(p)(a) \ .$$

Remark 3. For each abstract process net w.r.t. an equivalence relation $\sim \subseteq A \times A$ of a high-level Petri net there is a corresponding process net of the abstract high-level Petri net, and vice versa.

So we reduced abstract process nets to process nets of high-level Petri nets, and these to process nets of place/transition nets. Therefore the results of the previous sections can be applied to high-level Petri nets and abstract process nets as well.

An extreme case is to identify all elements of the domain, i.e., taking $\sim = A \times A$. In this case there is only one token type. In fact, this abstract high-level Petri net, the *skeleton* of the high-level Petri net, is not high-level any more; it can be viewed as a place/transition net. Its processes only provide information about the number of tokens used by transition occurrences ([6]). In our setting, this abstraction from individuality of all tokens is only useful if every transition occurrence can always happen with arbitrary tokens from the input places.

To apply the concept of abstract process nets, one has to define an equivalence relation on the domain. Often a suitable relation is known from the application, as it is the case with the letter example given above. Another way to obtain a suitable equivalence is to find isomorphic process nets and then to identify the respective used elements of the domain. A third way is to employ the syntactic representation of the input- and output mappings given e.g. by predicate/transition nets.

We are presently working on algorithms that calculate the coarsest equivalence relation for a given high-level net such that for any two equivalent tokens we have: for each process moving one of these tokens there is an isomorphic process moving the other one, and vice versa. This problem turns out to be a quite complex problem in general. Another present research direction is to find out whether a given equivalence relation satisfies the above property, i.e., whether the decomposition of tokens in two parts (called visible and invisible above), together with the assumption that the second part has no influence on the behavior, is justified.

7 Conclusions

We presented an approach for validating Petri net models by generation and analysis of causal runs, formally presented by process nets. The approach can be applied to both place/transition and high-level Petri nets. The particular emphasis of this paper is on the analysis of process nets with respect to specified properties, namely facts and goals, which is relevant for the analysis of the modeled system and on a step-wise system design where process nets that do not satisfy a given specification are identified (and possibly) ruled out for validation. A crucial aspect of process net generation is the termination of process nets.

The concept is partly implemented in the *VIPtool*, see
`http://www.informatik.ku-eichstaett.de/projekte/vip/` .

References

1. Best, E., C. Fernandez C.: *Nonsequential Processes*, Springer-Verlag, Berlin Heidelberg New York (1988) 28, 31
2. Desel, J.: Basic Linear Algebraic Techniques for Place/Transition Nets. In: Reisig, W., Rozenberg, G. (eds.): *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science, Vol. 1491. Springer-Verlag, Berlin Heidelberg New York (1998) 257–308 27
3. Desel, J.: Validation of Process Models by Construction of Process Nets. In: van der Aalst, W., Desel, J., Oberweis, A. (eds.): *Business Process Management*, Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin Heidelberg New York (2000) 110–128 23, 35
4. Desel, J., Erwin, T.: Modeling, Simulation and Analysis of Business Processes. In: van der Aalst, W., Desel, J., Oberweis, A. (eds.): *Business Process Management*, Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin Heidelberg New York (2000) 129–141 23
5. Desel, J., Juhás, G.: "What is a Petri Net?" Informal Answers for the Informed Reader. In: Ehrig, H., Juhás, G., Padberg, J., Rozenberg, G. (eds.): *Unifying Petri Nets*, Lecture Notes in Computer Science, Vol. 2128. Springer-Verlag, Berlin Heidelberg New York (2001) 1–25 23
6. Desel, J., Kindler, E..: Petri Nets and Components – Extending the DAWN Approach?. In: Moldt, D. (ed.): *Workshop on Modelling of Objects, Components. and Agents*, Aarhus, Denmark, DAIMI PB–553 (2001) 21–36 41
7. Desel, J., Reisig, W.: Place/Transition Petri Nets. In: Reisig, W., Rozenberg, G. (eds.): *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science, Vol. 1491. Springer-Verlag, Berlin Heidelberg New York (1998) 122–173 27, 29, 35
8. Ehrig, H., Hoffmann, K., Padberg, J., Baldan, P., Heckel, R.: High-Level Net Processes. In: Brauer, W., Ehrig, H., Karhumäki, J., Salomaa, A. (eds.): *Formal and Natural Computing – Essays Dedicated to Grzegorz Rozenberg*, Lecture Notes in Computer Science, Vol. 2300. Springer-Verlag, Berlin Heidelberg New York (2002) 191–219 39, 40
9. Engels, G., Heckel, R., Sauer, S.: UML – A Universal Modeling Language? In: Nielsen, S., Simpson, D. (eds.): *Application and Theory of Petri Nets 2000*, Lecture Notes in Computer Science, Vol. 1825. Springer-Verlag, Berlin Heidelberg New York (2000) 24–38 24

10. Esparza, J.: Model Checking Using Net Unfoldings. *Science of Computer Programming* Vol. 23 (1994) 151–195 28, 33, 34
11. Genrich, H., Thieler-Mevissen, G.: The Calculus of Facts. *Mathematical Foundations of Computer Science*, Springer-Verlag, Berlin Heidelberg New York (1976) 588–595 27, 35
12. Goltz, U., Reisig, W.: The Non-Sequantial Behaviour of Petri Nets. *Information and Computation* Vol. 57 (1983) 125–147 29
13. Jensen, K.: *Coloured Petri Nets, Vol.1: Basic Concepts*. 2nd edition, Springer-Verlag, Berlin Heidelberg New York (1995) 37
14. Reisig, W.: *A Primer in Petri Net Design*, Springer-Verlag, Berlin Heidelberg New York (1992) 31, 32
15. Reisig, W.: *Elements of Distributed Algorithms*, Springer-Verlag, Berlin Heidelberg New York (1995) New York (1995) 27, 33, 37
16. Verbeek, H. M. W., Basten, T., van der Aalst, W. M. P.: Diagnosing Workflow Processes using Woflan, *The Computer Journal* Vol. 44(4) (2001) 246–279 27

The Real-Time Refinement Calculus: A Foundation for Machine-Independent Real-Time Programming

Ian J. Hayes

School of Information Technology and Electrical Engineering
The University of Queensland, Brisbane, 4072, Australia
ianh@itee.uq.edu.au

Abstract. The real-time refinement calculus is an extension of the standard refinement calculus in which programs are developed from a pre-condition plus post-condition style of specification. In addition to adapting standard refinement rules to be valid in the real-time context, specific rules are required for the timing constructs such as delays and deadlines. Because many real-time programs may be nonterminating, a further extension is to allow nonterminating repetitions.
A real-time specification constrains not only what values should be output, but when they should be output. Hence for a program to implement such a specification, it must guarantee to output values by the specified times. With standard programming languages such guarantees cannot be made without taking into account the timing characteristics of the implementation of the program on a particular machine. To avoid having to consider such details during the refinement process, we have extended our real-time programming language with a deadline command. The deadline command takes no time to execute and always guarantees to meet the specified time; if the deadline has already passed the deadline command is infeasible (miraculous in Dijkstra's terminology).When such a real-time program is compiled for a particular machine, one needs to ensure that all execution paths leading to a deadline are guaranteed to reach it by the specified time. We consider this checking as part of an extended compilation phase. The addition of the deadline command restores for the real-time language the advantage of machine independence enjoyed by non-real-time programming languages.

1 Introduction

This paper is intended to summarise and highlight the main conceptual advances made in developing a refinement calculus for real-time programs. The overriding breakthrough is the development of a machine-independent approach to real-time program refinement by extending the real-time programming language with a deadline command that allows machine-independent real-time programs to be expressed.

Because the real-time refinement calculus is based on the standard refinement calculus, we begin in Sect. 2 with an overview of some of the important concepts

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 44–58, 2002.
© Springer-Verlag Berlin Heidelberg 2002

from the refinement calculus. Sect. 3 introduces the machine-independent approach to real-time programming, and primitive real-time commands including the deadline command. Real-time refinement laws are introduced in Sect. 4; for the standard laws to be applicable to real-time programs the laws need to be constrained using notions such as idle invariance. Because real-time programs commonly contain nonterminating processes, Sect. 5 extends the calculus to handle these.

2 The Refinement Calculus

The goal of the refinement calculus is to provide formal support for the stepwise refinement of programs from specifications [15]. The primary innovation is to extend Dijkstra's weakest precondition calculus [2] with a specification command [1,14] that allows specifications and programs to be treated within a single framework: a wide-spectrum programming language with a subset that corresponds to executable *code*. The specification command

$$x{:}\left[P,\quad Q\right],$$

if executed in a state in which the precondition P holds, guarantees to terminate in a state in which the postcondition Q holds, and only modify variables in the frame x. If the precondition P is omitted, it is assumed to be *true*. Such a command is not considered executable, in fact, it does not satisfy Dijkstra's law of the excluded miracle because a specification such as $:\left[\mathit{true},\quad \mathit{false}\right]$ when started in any state (precondition *true*) guarantees to achieve the postcondition *false* (from which any other postcondition follows). This specification is called **magic**.

Special cases of the specification command are the assumption (or assertion) command and the guard (or coercion) command

$$\begin{aligned}\left\{P\right\} &\mathrel{\widehat{=}} :\left[P,\quad \mathit{true}\right]\\ \left[B\right] &\mathrel{\widehat{=}} :\left[\mathit{true},\quad B\right]\end{aligned}$$

The assumption command records assumptions made by the program: it has no effect if P holds but is equivalent to **abort** otherwise. The guard command has no effect if B holds but is equivalent to **magic** otherwise because it is infeasible.

3 Machine Independence and Real Time

The most significant advantage of programming languages over assembly languages is machine independence. However, for real-time programs written in standard programming languages the execution time of (paths through) a program depends on the particular implementation, and hence the real-time characteristics of these languages are machine dependent.

The deadline command. To overcome this problem we have introduced a command of the form **deadline** D, which guarantees to complete by absolute time D and takes no time to execute [10,3]. If a deadline command is executed at a time strictly after its specified deadline, then it is miraculous. Clearly such a command cannot be implemented in the normal way by generating machine code. Instead it is 'implemented' by ensuring that all program paths ending in a deadline are guaranteed to reach the deadline before it expires. For example, the program

$\{\tau \leq s\};$
$C;$ -- command that does not modify s
deadline $s + 1\,\text{ms}$

in which τ stands for the current time, has an assumption that it is executed before time s and has a deadline of $s + 1\,\text{ms}$. Hence if the command C can always be executed in less than one millisecond, the deadline will always be reached on time. The special variable τ is of type $Time_\infty$, where $Time_\infty$ is the set of nonnegative reals plus infinity. In addition to giving the sets of values for types, we also give their dimensions of measurement [9]; in this case the dimension is time.

Timing path analysis. The inclusion of the deadline command allows one to write machine-independent real-time programs, and this in turn allows the use of a machine-independent real-time refinement calculus, in which a real-time specification (with hard timing constraints) can be refined to a real-time program (with deadline commands). Before such a program can be used on a particular target machine, it must be checked to ensure that all deadlines will be reached on time [4]. This timing analysis is the price to be paid for machine-independence in the earlier phases. However, to verify that a real-time program meets its real-time obligations, such an analysis of the final program is necessary anyway. Fig. 1 gives an overview of the program development approach.

Defining the deadline command. Real-time commands, such as the deadline command, and commands to get the current time, read an input, and delay until a given time, can be defined in terms of specification commands, in which the special variable τ, stands for the current time and is implicitly in the frame. Within the postcondition, τ stands for the finish time of the command and τ_0 for the start time.

$$\textbf{deadline}\, D \mathrel{\widehat{=}} \left[\tau_0 = \tau \wedge \tau \leq D\right]$$

If the time at which a deadline command is 'executed' is strictly after D, then the deadline command is miraculous (will achieve any postcondition). To ensure it is not miraculous for a particular implementation, we need to carry out the timing analysis indicated above.

Defining a command to get the current time. The command t : **gettime** assigns t an approximation to the current time. In our syntax all variables modified by

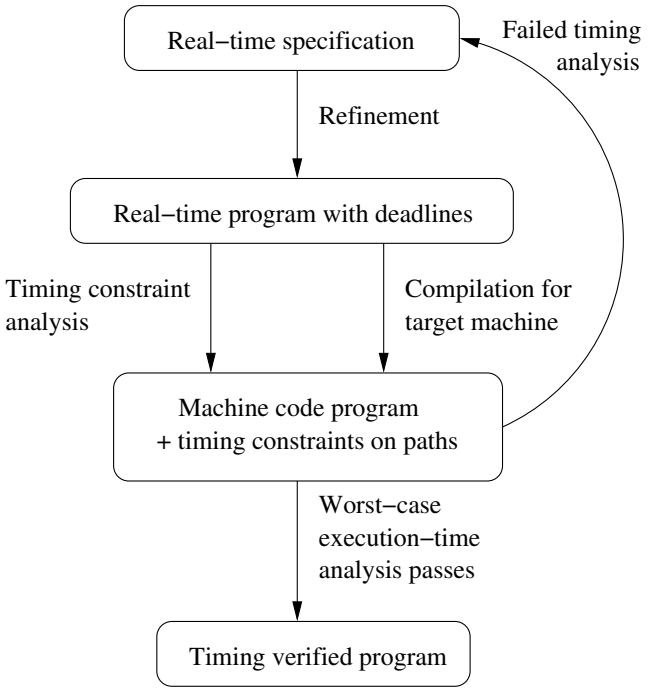


Fig. 1. Development process for real-time programs

a command are to the left of a colon at the beginning of the command. An implementation cannot accurately represent all real numbers (as in type *Time*) and hence the type of t will be some subset of *Time*. A simple definition of the **gettime** command is

$$t : \mathbf{gettime} \mathrel{\widehat{=}} t{:}\left[\tau_0 \leq t \leq \tau\right]$$

This assumes that the clock resolution is fine enough that there is always a clock tick during the execution of the command. If this is not the case (e.g., the clock resolution is 20 ms) then the following more complicated definition is appropriate.

$$t : \mathbf{gettime} \mathrel{\widehat{=}} t{:}\left[\tau_0 - \mathit{resolution} \leq t \leq \tau\right]$$

Even this is insufficient because the real numbers are unbounded. To solve this problem a precondition can be added to the **gettime** command that assumes the time is before the implementation imposed limit on the maximum time.

Defining a read from an external input. We assume the value of an external input is under the control of the environment and not the program. For real-

time programs we would like to consider not only discrete inputs but also analog inputs (sampled by an analog-to-digital converter). Hence we model an external input (or output) as a function from *Time* to the type of the input. A simple definition of a read command assumes that the value read is the value of the input at some time during the execution of the read command:

$$x : \mathbf{read}(i) \mathrel{\widehat{=}} x{:}\, \big[\exists\, t : [\tau_0 \mathinner{..} \tau] \bullet x = i(t)\big]$$

This definition is not completely accurate. For an analog input a sample is taken over a (small) time interval. Hence there may be an error in the sampled value as well. For a discrete value there is the further problem that if the input is changing from one value x to another value y during the execution of the read command, the value read may be neither x nor y. We handle this problem by saying that the value of a changing input may be any possible value of its type (even that constrains the implementation). Hence a read of a discrete value is only really useful if there is some form of synchronisation that guarantees that its value is not changing while the read takes place. For boolean values we make the stronger assumption that the change in the boolean is effectively instantaneous; that is, there is only a single transition (or no transitions at all if its value does not change).

Defining the delay command. The delay command waits until absolute time D as evaluated at the start time of the delay command, τ_0. We use the notation $D \mathbin{@} \tau_0$ to stand for the value of the expression D at time τ_0.

$$\mathbf{delay\ until}\, D \mathrel{\widehat{=}} \big[D \mathbin{@} \tau_0 \leq \tau\big]$$

A delay is straightforward to implement. Note that, although we require the delay command to terminate, we put no upper bound on the termination time of the delay command (the programmer may however specify an upper bound by following the delay command with a deadline command).

Idle-stable expressions. As the delay command may take time to execute (unlike the deadline command) we restrict the allowable expressions for the delay time D to *idle-stable* expressions: expressions that do not change during the execution of the command. The main problem here is with expressions that refer to the current time or to the value of external inputs (whose values may change independently over time). An expression D is *idle-stable* provided for all times τ_0 and τ,

$$\tau_0 \leq \tau < \infty \wedge \mathit{stable}(\mathit{outputs}, [\tau_0 \mathinner{..} \tau]) \Rrightarrow D \mathbin{@} \tau_0 = D \mathbin{@} \tau$$

where *outputs* is the set of all the externally visible outputs of the program, $\mathit{stable}(\mathit{outputs}, [\tau_0 \mathinner{..} \tau])$ means that these outputs do not change over the closed time interval from τ_0 to τ, and '$\Rrightarrow$' stands for implication for all possible values of the variables. References to local variables (as opposed to external inputs and outputs) in the expressions $D \mathbin{@} \tau_0$ and $D \mathbin{@} \tau$ refer to the same variable to represent the fact that local variables are unchanged over an idle period.

4 Real-Time Refinement Laws

In the real-time refinement calculus we would like to have refinement laws that mirror the laws of the standard calculus as closely as possible. For example, consider the following refinement law which refines a specification command to an *if* command with a specification command in each branch. We use the notation $S \sqsubseteq T$ to mean S is refined by (or implemented by) T.

$$x{:}\,\big[P,\quad Q\big] \sqsubseteq \textbf{if}\, B\, \textbf{then}\, x{:}\,\big[P \wedge B,\quad Q\big]\, \textbf{else}\, x{:}\,\big[P \wedge \neg\, B,\quad Q\big]$$

The specification in the 'then' branch can assume both P and B, and the 'else' branch can assume both P and not B. Both branches must establish Q.

Unfortunately this law is not valid for real-time programs. For example, if the precondition P contains $\tau \leq D$ (e.g., it was preceded by **deadline** D), it is not necessarily the case that P will still hold after the condition B is evaluated, because the evaluation of B takes time. To allow the above law to be used for the real-time language, we introduce constraints on the form of B, P, and Q. As for the delay command, we require that B is *idle-stable*, so that its value is not changing while it is being evaluated.

Idle invariance. For the precondition P we need a slightly different constraint. It should be *idle-invariant*: if it holds at a time τ_0 then it should hold at any later time τ, provided none of the variables under the control of the program change. That is,

$$\tau_0 \leq \tau < \infty \wedge \mathit{stable}(\mathit{outputs}, [\tau_0 \mathinner{..} \tau]) \wedge P \mathbin{@} \tau_0 \Rrightarrow P \mathbin{@} \tau.$$

For example, the predicate $\tau \leq D$ is not idle-invariant for the reasons discussed above, but the predicate $D \leq \tau$ is (assuming D itself is an idle-invariant expression). Note that the predicate $D \leq \tau$ may be false at time τ_0 but true at the later time τ. This satisfies the requirements for it to be idle-invariant, but not for it to be idle-stable; for the latter the value may not change.

An idle-invariant predicate P, which holds before the evaluation of the condition, B, of the *if* command, will still hold after the evaluation of the guard, i.e., at the start of the 'then' and 'else' branches of the command.

Pre- and post-idle-invariant. For the postcondition, Q, in the specification in the above law we need some similar constraints. In the postcondition Q of the specification command on the left side of the refinement, τ_0 refers to the start time of the *if* command and τ to the finish time of the *if* command. However, these times may differ from the start and finish times of the commands in the two branches of the *if* command, due to the time taken to evaluate the condition and select one of the two branches, and the time taken to exit from a branch once it has completed. Hence we need Q to be impervious to these 'idle' periods in which decisions are being made, but the program is not changing any of its variables. As with idle-invariance, the problems arise from references to the current time

variable, τ, and to external input variables. A predicate Q is pre-idle-invariant if for a fresh time-valued variable u,

$$u \leq \tau_0 \leq \tau \wedge \tau_0 < \infty \wedge \mathit{stable}(\mathit{outputs}, [u \mathinner{..} \tau_0]) \wedge Q \Rrightarrow Q\,[\tau_0 \backslash u]$$

and Q is post-idle-invariant if

$$\tau_0 \leq \tau \leq u < \infty \wedge \mathit{stable}(\mathit{outputs}, [\tau \mathinner{..} u]) \wedge Q \Rrightarrow Q\,[\tau \backslash u]\,.$$

The overall restrictions on the predicates for the refinement law to introduce an *if* command above are that B is idle-stable, P is idle-invariant, and Q is both pre- and post-idle-invariant. The laws for introducing other control structures such as repetitions, procedure calls, and introducing local variable blocks all make use of similar restrictions.

5 Nonterminating Processes

5.1 Specification of Nonterminating Processes

It is common for real-time systems to contain processes that monitor inputs and control outputs. Conceptually such processes may execute forever, that is, if the machine executes without fail forever, so should the process. Hence another extension in the real-time calculus is to allow specifications of nonterminating processes and refinement to a nonterminating repetition. To allow for the specification of nonterminating processes, we allow the special time variable τ to take on the value infinity to indicate nontermination, and introduce a possibly nonterminating variant of the specification command:

$$\infty\ x{:}\,[P, \quad Q]$$

The earlier terminating specification command can be defined as a special case of this.

$$x{:}\,[P, \quad Q] \mathrel{\widehat{=}} \infty\ x{:}\,[P, \quad Q \wedge \tau < \infty]$$

Example: smoke detector. We give an example specification that we eventually develop into a nonterminating repetition. A process is required to monitor a smoke detector input and when smoke is detected terminate within p seconds. This would normally be followed by other actions such as raising an alarm, but we do not consider those here. If the smoke detector never triggers, the process never terminates. The smoke detector is modelled by a single boolean input, *smoke*. If *smoke* triggers then it is assumed that it will stay triggered for at least a period of p seconds, where p is some finite constant. We use the time *trigger* to stand for the exact time at which *smoke* triggers; if it never triggers then *trigger* is infinity. The input *smoke* is *false* up until time *trigger*, and if *trigger* is not infinity *smoke* is true for at least another p seconds:

$$\left\{\begin{array}{l} \forall\, t : \mathit{Time} \bullet (t < \mathit{trigger} \Rightarrow \neg\ \mathit{smoke}(t)) \wedge \\ \qquad\qquad (\mathit{trigger} \leq t \leq \mathit{trigger} + p \Rightarrow \mathit{smoke}(t)) \end{array}\right\} \tag{1}$$

If we explicitly separate the terminating and nonterminating cases, the process can be specified as follows.

$$\infty \left[\tau \leq \mathit{trigger}, \quad \begin{array}{l}(\tau < \infty \wedge \mathit{trigger} \leq \tau \leq \mathit{trigger} + p) \vee \\ (\tau = \infty \wedge \mathit{trigger} = \infty)\end{array}\right] \tag{2}$$

5.2 Sequential Composition

Standard terminating sequential composition. Although all the compound commands are affected by the possibility of nontermination (of components), the interesting commands are sequential composition and repetition. Our laws for sequential composition are in a style similar to that of Jones [13] in which postconditions are treated as relations and may be composed. In the following law P_0, P_1 and P_2 are predicates on a single state (no zero-subscripted variables are allowed), and R_1 and R_2 are predicates on both the before (zero-subscripted) and after (unsubscripted) states. P_0 is assumed to hold initially and P_2 should hold on termination of the whole sequential composition; P_1 is used as the intermediate condition which should be established by the first command and hence assumed to hold initially by the second command. R_1 and R_2 effectively define relations between states, and we use the notation $R_1 \fatsemi R_2$ to stand for their relational composition: a before state is related to a final state by $R_1 \fatsemi R_2$ if there exists an intermediate state related to the before state by R_1 and which is related to the after state by R_2. The standard law for sequential composition is as follows.

$$x{:}\left[P_0, \quad P_2 \wedge (R_1 \fatsemi R_2)\right] \sqsubseteq x{:}\left[P_0, \quad P_1 \wedge R_1\right] ; \; x{:}\left[P_1, \quad P_2 \wedge R_2\right]$$

Real-time possibly nonterminating sequential composition. For the real-time case, a sequential composition may not terminate either because its first command does not terminate, or because its first command terminates but its second does not. Our law for sequential composition reflects these possibilities. Each of the two commands being composed may either terminate and establish $P_1 \wedge R_1$ and $P_2 \wedge R_2$, respectively, as for the standard law, or they may fail to terminate. In the latter case if the first command fails to terminate it establishes the predicate Q_1, and if the second command fails to terminate it establishes Q_2. If both commands terminate then the composition terminates and establishes P_2 and the relational composition of R_1 and R_2. If the first command fails to terminate, it establishes Q_1, and hence so does the sequential composition. If the first command terminates but the second does not, then the sequential composition establishes $R_1 \fatsemi Q_2$.

$$\begin{array}{l} \infty\, x{:}\left[P_0, \quad (\tau < \infty \wedge P_2 \wedge (R_1 \fatsemi R_2)) \vee (\tau = \infty \wedge (Q_1 \vee (R_1 \fatsemi Q_2))\right] \\ \sqsubseteq \\ \infty\, x{:}\left[P_0, \quad (\tau < \infty \wedge P_1 \wedge R_1) \vee (\tau = \infty \wedge Q_1)\right] ; \\ \infty\, x{:}\left[P_1, \quad (\tau < \infty \wedge P_2 \wedge R_2) \vee (\tau = \infty \wedge Q_2)\right] \end{array}$$

This law generalises that of Jones. His law is the special case when both Q_1 and Q_2 are the predicate *false*. For the predicates Q_1 and Q_2 used for the non-terminating cases it makes no sense for them to constrain the final state (there is no final state). Instead they relate the external inputs and outputs over time—recall that the inputs and outputs are modelled as functions of time.

Example: separating the deadline. The postcondition of the specification (2) contains a constraint, $\tau \leq trigger + p$, on the final time for the terminating case. We can use the law for sequential composition to separate out this deadline; it becomes condition P_2 in the law. The remaining predicates are as follows: P_0 is $\tau \leq trigger$; P_1 is just *true*; R_1 is $trigger \leq \tau$; R_2 is $\tau_0 = \tau$; Q_1 is $trigger = \infty$; and Q_2 is *false*. The instantiation of the law is as follows; note that the specification being refined is equivalent to (2).

$$
\begin{aligned}
&\infty \left[\tau \leq trigger,\ \begin{array}{l}(\tau < \infty \land \tau \leq trigger + p \land ((trigger \leq \tau) \fatsemi (\tau_0 = \tau))) \lor \\ (\tau = \infty \land trigger = \infty \lor ((trigger \leq \tau) \fatsemi (false)))\end{array}\right] \\
\sqsubseteq \\
&\infty \left[\tau \leq trigger,\ \begin{array}{l}(\tau < \infty \land true \land trigger \leq \tau) \lor \\ (\tau = \infty \land trigger = \infty)\end{array}\right]; \qquad (3) \\
&\infty \left[true,\ \begin{array}{l}(\tau < \infty \land \tau \leq trigger + p \land \tau_0 = \tau) \lor \\ (\tau = \infty \land false)\end{array}\right] \qquad (4)
\end{aligned}
$$

The final specification (4) is equivalent to

$$\left[\tau_0 = \tau \land \tau \leq trigger + p\right]$$

which is equivalent to the command **deadline** $trigger + p$. Note that if the first command (3) never terminates, the second (deadline) command is never reached and hence its deadline only needs to be met when the first command terminates. Separating a deadline is a common strategy for refinement, and a specialised law for this situation has been developed [11]. The remaining specification (3) is refined to a repetition below.

5.3 Repetition

Standard terminating repetition law. In the standard refinement calculus a non-terminating repetition is of no use, and hence the standard rule for introducing a repetition makes use of either a variant or a well-founded relation to ensure that the repetition terminates. The standard law (here based on Jones's approach [13]) makes use of an invariant I and a well-founded relation R. The invariant is assumed to hold initially, and is maintained by the body of the repetition. Hence I holds on termination of the repetition, along with the negation of the guard of the repetition. The body of the repetition also ensures that the relation R holds between the before and after states on each iteration. Hence the transitive closure, R^*, of the relation R holds between the before and after states of the whole repetition. R being well-founded ensures that there cannot

be an infinite sequence of successively related states, and hence guarantees that the repetition terminates.

$$x{:}\left[I, \quad \neg B \wedge I \wedge R^*\right] \sqsubseteq \mathbf{do}\, B \rightarrow x{:}\left[B \wedge I, \quad I \wedge R\right] \mathbf{od}$$

Nonterminating repetition. In the real-time language a repetition may sample an input and control an output forever. Hence a nonterminating repetition can have a useful effect. Note that for a nonterminating repetition there is no final state, and hence its effect is expressed not in terms of a predicate on the state of the program at a single instant, but as its effect on the inputs and outputs over time.

Here we restrict ourselves to a repetition whose body is guaranteed to terminate on every iteration. The more general case is treated in [8,6]. If the whole repetition terminates then, as for the standard law, on termination I holds and the guard B does not and R^* holds between the before and after states of the whole repetition. The nonterminating case is more complex. There is no concept of a final state for a nonterminating computation. Hence the effect of a nonterminating repetition consists of the combination of the partial effects of each iteration. Because we have chosen the case in which each iteration terminates, for the whole repetition not to terminate, there must be an infinite number of iterations of the body of the repetition.

Infinite iteration. To handle this we introduce the notion of infinite iteration of a relation R. This is a relation composed with itself infinitely many times. To define this we make use of infinite sequences $\vec{t}$ and $\vec{v}$ corresponding to the sequences of intermediate times and variable values between each composition of the relation. The initial time and variable values in the sequence, i.e., $\vec{t}(0)$ and $\vec{v}(0)$, correspond to the start time of the repetition τ_0 and the initial value of the program variables, represented here by v_0 (of type T_v). Successive values of times and variable values in the sequence are related by R.

$$R^\infty \mathrel{\widehat{=}} (\exists\, \vec{t} : \mathbb{N} \rightarrow \mathit{Time};\ \vec{v} : \mathbb{N} \rightarrow T_v \bullet \vec{t}(0) = \tau_0 \wedge \vec{v}(0) = v_0 \wedge (\forall\, i : \mathbb{N} \bullet R \left[\frac{\vec{t}(i), \vec{t}(i+1), \vec{v}(i), \vec{v}(i+1)}{\tau_0, \tau, v_0, v}\right]))$$

Nonterminating repetition law. The law for introducing a repetition assumes the following are given: an idle-stable, boolean-valued expression, B—the guard; a single-state, idle-invariant predicate, I—the loop invariant; a pre-idle-invariant relation R'; and a weaker pre- and post-idle-invariant relation R. The body of the repetition assumes that the guard B and the invariant I hold initially, and it re-establishes I and establishes the relation R', and hence R, between the before and after states of the body. The relation R' is not required to be post-idle-invariant. It is used for reasoning about the nonterminating case, whereas the weaker pre- and post-idle-invariant relation R is used for the terminating case.

The terminating case of the law is essentially the same as the standard law above. For the nontermination case the body of the repetition is executed infinitely many times, and hence the overall effect is that of the infinite iteration

of the body. In the infinite iteration we include not only the relation R', but also the fact that B and I hold at the beginning of each iteration. (B and I also hold at the end of each iteration, but adding that would be redundant because the state at the end of one iteration is same as the state at the start of the next.) In addition, because we are modelling a repetition in which each iteration takes some minimal time d (where d is a fresh name), we have included the constraint that each iteration takes some minimum time d.

$$\infty\, x{:}\left[\begin{array}{ll} & (\tau < \infty \wedge \neg\, B \mathbin{@} \tau \wedge I \wedge R^*) \vee \\ I, & (\tau = \infty \wedge (\exists\, d : \mathit{Time} \bullet 0 < d \wedge \\ & \qquad (\tau_0 + d \leq \tau \wedge B \mathbin{@} \tau_0 \wedge I \mathbin{@} \tau_0 \wedge R')^\infty)) \end{array}\right]$$
$$\sqsubseteq$$
$$\mathbf{do}\, B \rightarrow x{:}\left[B \mathbin{@} \tau \wedge I, \quad I \wedge R'\right]\, \mathbf{od}$$

Note that the same value of d is used for all iterations. This rules out Zeno-like behaviour—in which each iteration takes half the time of the previous iteration—and hence ensures that the infinite iteration encompasses the whole of time up to infinity and not just some finite range of time.

If R is well founded then R^∞ is false, and the law reduces to that of Jones above.

Example: a nonterminating repetition. In order to develop a repetition that refines the specification (3), we need to find a guard B, an invariant I, and relation R' and a weaker relation R, such that both of the following hold.

$$\neg\, B \mathbin{@} \tau \wedge I \wedge R^* \Rightarrow \mathit{trigger} \leq \tau \tag{5}$$
$$\left(\begin{array}{l} \exists\, d : \mathit{Time} \bullet 0 < d \wedge \\ \quad (\tau_0 + d \leq \tau \wedge B \mathbin{@} \tau_0 \wedge I \mathbin{@} \tau_0 \wedge R')^\infty \end{array}\right) \Rightarrow \mathit{trigger} = \infty \tag{6}$$

We introduce a boolean variable tr, which is used to sample the value of *smoke*. If tr is true then the time is after *trigger*; hence the invariant has as a conjunct

$$tr \Rightarrow \mathit{trigger} \leq \tau$$

If the repetition has a guard of $\neg\, tr$, the negation of the guard and this invariant are enough to establish (5). For this example R is just *true*.

If tr is false then *trigger* is later than the time of the most recent sample. We introduce an auxiliary variable *sample* to stand for (a lower bound on) this time. (Auxiliary variables are used purely for reasoning about the timing behaviour of programs and no code needs to be generated for assignments to them [5].) The invariant includes the conjunct

$$\neg\, tr \Rightarrow \mathit{sample} \leq \mathit{trigger}$$

Both conjuncts of the invariant are established by an initial assignment before the repetition consisting of

$$\mathit{sample}, tr := \tau, \mathit{false}$$

Because the smoke detector is only guaranteed to stay *true* for p seconds, it must be sampled within p seconds of the previous sample in order not to be missed. The relation R' puts an upper bound on the time of completion of the body of the repetition in terms of the time of the previous sample, $sample_0$, i.e., the value of *sample* at the start of an iteration.

$$R' \mathrel{\widehat{=}} \tau \leq sample_0 + p$$

If we combine $B \mathbin{@} \tau_0 \wedge I \mathbin{@} \tau_0 \wedge R'$ we get

$$\neg\, tr_0 \wedge sample_0 \leq trigger \wedge \tau \leq sample_0 + p$$

This implies $\tau \leq trigger + p$, and hence the infinite iteration in the postcondition of the law above implies

$$\vec{t}(0) = \tau_0 \wedge (\forall\, i : \mathbb{N} \bullet \vec{t}(i) + d \leq \vec{t}(i+1) \wedge \vec{t}(i+1) \leq trigger + p)$$

where d is a strictly positive time. The minimum separation between successive elements of $\vec{t}$ guarantees that its values are unbounded, and because $trigger + p$ is greater than all elements of $\vec{t}$ (and p is a finite constant), that *trigger* must be infinity. Hence condition (6) holds.

The refined program. The code developed so far is

$$\begin{array}{l}
\{\tau \leq trigger\}; \\
sample, tr := \tau, false; \\
\textbf{do}\ \neg\, tr \rightarrow \\
\qquad tr, sample{:}\ [\neg\, tr \wedge I, \quad I \wedge \tau \leq sample_0 + p] \qquad (7) \\
\textbf{od}; \\
\textbf{deadline}\ trigger + p
\end{array}$$

where $I \mathrel{\widehat{=}} (tr \Rightarrow trigger \leq \tau) \wedge (\neg\, tr \Rightarrow sample \leq trigger)$. The body (7) of the repetition can be refined to the following code by introducing an additional auxiliary variable *prev* to capture the time of the previous sample (see Fig. 2). We have added the labels (*A*) to (*D*) to facilitate discussion of timing analysis.

$$\begin{array}{l}
\text{A} :: \{\tau \leq trigger\}; \\
\quad sample, tr := \tau, false; \\
\quad \textbf{do}\ \neg\, tr \rightarrow \\
\quad\ \text{B} :: prev, sample := sample, \tau; \\
\qquad tr : \textbf{read}(smoke); \\
\quad\ \text{C} :: \textbf{deadline}\ prev + p; \\
\qquad \{(tr \Rightarrow trigger \leq \tau) \wedge (\neg\, tr \Rightarrow sample \leq trigger)\} \\
\quad \textbf{od}; \\
\text{D} :: \textbf{deadline}\ trigger + p
\end{array}$$

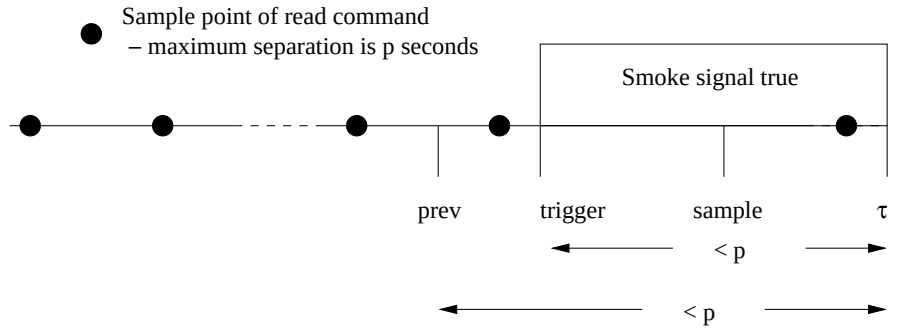


Fig. 2. Sampling of smoke detector

Timing constraint analysis. The above is a machine-inpendent real-time program which contains deadline commands to indicate the timing constraints on paths through the program. The first path we consider starts at the beginning of the program (A), assigns *sample* and *tr*, enters the repetition, assigns *prev* and *sample*, and reads *smoke* into *tr*, before reaching the deadline (C) of $prev + p$. It has a start time of *sample* and a deadline of $prev + p$, but in this case the value assigned to *prev* is *sample* and hence the deadline expressed in terms of the initial values of the program variables is $sample + p$. Therefore provided the path executes in a time less than or equal to p, the deadline will always be met.

For non-initial iterations of the repetition, we consider the path that starts at (B), executes the body of the repetition, restarts the repetition because the guard holds, and executes down to the deadline (C). The start time of the path is captured initially in *sample* and the deadline at the end is $prev + p$, but in this case on the start of the second execution of the body *prev* is assigned *sample*, and hence the deadline (in terms of the initial values of the program variables) is also $sample + p$ and the overall constraint on the execution time of the path is p.

The next path we consider is the same as the previous path, but then exits the repetition because *tr* holds. The start time of the path is *sample* and the deadline is $trigger + p$, but at the end of the first execution of the body the invariant holds and *tr* is false (because the iteration repeats). Hence $sample \leq trigger$ initially and the constraint on this path is also p. There is one final path that starts at (A), enters the repetition, and exits after one iteration. It has the same constraint.

The process of analysing the timing paths of a program is nontrivial. As we saw in this example reasoning about a timing path can rely on information from a loop invariant. In general the process is incomputable, but for sufficiently restricted programs the analysis can be automated [4].

6 Conclusions

The real-time refinement calculus provides a suitable theoretical basis for the stepwise development of real-time programs from specifications in a largely machine-independent fashion. The deadline command was introduced to allow real-time programs to be expressed independently of the target machine. In order to promote the standard refinement calculus to the real-time framework one needs to be careful about assertions that refer to time or to the values of external inputs, whose values may change over time independently of the action of the program. Hence expressions within real-time commands are restricted to be idle-stable, and for refinement laws single-state predicates are restricted to be idle-invariant and before-after state predicates (relations) are restricted to be pre- and post-idle-invariant. A predicative semantics for the calculus has been developed [7].

To support nonterminating real-time processes it is necessary to allow for nonterminating specifications and repetitions. The fact that commands may be nonterminating affects other refinement laws, in particular the refinement law for sequential composition has to be extended to allow for either component to not terminate. The law for a repetition not only needs to handle the terminating case but also the nonterminating case. The latter may be either due to a repetition that has a terminating body but whose guard never becomes false (as treated above), or due to a repetition that executes some finite number of iterations of the body but then on the last iteration the body does not terminate (see [8]). Current research is examining extending the calculus to handle concurrency and to handle timeouts.

Hooman and Van Roosmalen [12] have also developed a platform-independent approach similar to ours, although they use timing annotations on commands rather than treating deadlines as commands in their own right. The idle invariance properties discussed here could also be applied to their work, as could the more general approach to real-time repetitions.

Acknowledgements

This research was supported by Australian Research Council (ARC) Discovery Grant DP0209722, *Derivation and timing analysis of concurrent real-time software*. I would like to thank Robert Colvin for feedback on earlier drafts of this paper, and the members of IFIP Working Group 2.3 on Programming Methodology for feedback on this topic.

References

1. R.-J. Back. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980. 45
2. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 45

3. C. J. Fidge, I. J. Hayes, and G. Watson. The deadline command. *IEE Proceedings—Software*, 146(2):104–111, April 1999. 46
4. S. Grundon, I. J. Hayes, and C. J. Fidge. Timing constraint analysis. In C. McDonald, editor, *Computer Science '98: Proc. 21st Australasian Computer Sci. Conf. (ACSC'98),* Perth, 4–6 Feb., pages 575–586. Springer, 1998. 46, 56
5. I. J. Hayes. Real-time program refinement using auxiliary variables. In M. Joseph, editor, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Comp. Sci.*, pages 170–184. Springer, 2000. 54
6. I. J. Hayes. Reasoning about non-terminating loops using deadline commands. In R. Backhouse and J. N. Oliveira, editors, *Proc. Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 60–79. Springer, 2000. 53
7. I. J. Hayes. A predicative semantics for real-time refinement. In A. McIver and C. C. Morgan, editors, *Essays in Programming Methodology*. Springer, 2002. 26 pages. Accepted. 57
8. I. J. Hayes. Reasoning about real-time repetitions: Terminating and nonterminating. *Science of Computer Programming*, 2002. 32 pages. Accepted. 53, 57
9. I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3):329–347, 1995. 46
10. I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop (NFMW'96)*. Springer, 1997. 46
11. I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001. 52
12. J. Hooman and O. van Roosmalen. An approach to platform independent real-time programming: (1) formal description. *Real-Time Systems*, 19(1):61–85, July 2000. 57
13. C. B. Jones. Program specification and verification in VDM. Technical Report UMCS-86-10-5, Department of Computer Science, University of Manchester, 1986. 51, 52
14. C.C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3), July 1988. 45
15. N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14(4):221–227, 1971. 45

The Challenge of Object Orientation for the Analysis of Concurrent Systems

Charles Lakos

Computer Science Department, University of Adelaide
Adelaide, SA, 5005, Australia
Charles.Lakos@adelaide.edu.au

Abstract. There have been a number of proposals for the integration of object-oriented capabilities into the Petri Net formalism. These proposals have demonstrated benefits for modelling, simulation and code generation. However, little has been said about analysis, particularly the use of state space methods, which are so prominent in Petri Net research and application. This talk will consider the impact of object oriented capabilities on state space analysis.

1 The Impact of Object Orientation

Object orientation has had a pervasive influence on Software Engineering over at least the last decade as both researchers and practitioners have striven for software that is comprehensible, flexible and maintainable [27]. Central to object orientation is the use of abstraction to manage complexity, with the result that objects are characterised by encapsulation and information hiding. Inheritance has been used to support incremental development — both to structure the original software and to aid the evolution of software.

Object orientation has achieved extensive application. Object oriented frameworks have become *de rigeur* for building Graphical User Interfaces [18,39]. Design methodologies based on objects have evolved into the industry standard of UML [3,4,31]. Middleware has emerged to support the construction of transparently distributed applications, adopting objects as the paradigm for interaction [33]. Common solutions to problems of structuring object oriented systems have been codified as Design Patterns [9]. Research into software components and mobile agents trace their roots to object orientation [28,40].

This is not to say that the adoption of object orientation has been universally accepted or trouble-free. There has been numerous proposals for appropriate type systems, including the debate about covariant versus contravariant subtyping [2,6,26,29]. It has been observed that software systems often exhibit concerns, such as security, which cut across the class structure, and this has led to the development of Aspect-oriented programming [19]. Others have noted that classes are too coarse a structure to cater for different subjective perspectives, and instead they propose the use of Subject-oriented programming where each subject encapsulates a single coherent piece of functionality, and where

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 59–67, 2002.
© Springer-Verlag Berlin Heidelberg 2002

subjects are combined to form classes using subject compositors [11]. Some have sought to distance the notion of components from that of objects [34]. The work on Software Architecture has identified the fact that performance of a system is often determined by the large scale design choices rather than the detailed decisions [5,32].

Despite these concerns, the adoption of object orientation is still widespread. The above somewhat cursory (and no doubt biased) view of object orientation is taken as the backdrop for this talk, and provides the impetus for including object oriented capabilities in a formalism such as Petri Nets. There have been a number of such proposals [1] and these have demonstrated benefits for modelling, simulation and code generation. As yet however, there seems to have been little said about analysis, and particularly about the use of state space methods. It is this issue which is of concern in this talk. We will argue that the inclusion of object oriented capabilities, as well as leading to comprehensible, flexible and maintainable formal models, can contribute to the more efficient analysis of such models.

We focus on *state space methods* for analysis, since these have a long history of application in Petri Net research. Here, the formal specification of the system is used to explore the state space, i.e. the set of all possible states and the transitions between them. The desired property is then evaluated against the state space. State space methods are conceptually simpler than theorem proving, are more amenable to automation, do not require highly trained personnel, and have the desirable property of indicating counter examples (when the desired property is not satisfied) [35]. However, state space methods have the significant disadvantage of the *state explosion problem* — for complex systems the number of possible states and transitions can be so large as to preclude exhaustive exploration. In such cases, reduction techniques, such as *Symmetric Occurrence Graphs* [16], and *Stubborn Sets* [35], can be employed so that only part of the state space is examined without affecting the property under consideration. As a result, state-based methods are still seen as one of the most promising formal reasoning techniques [35]. It is the goal of this talk to explore the effect of object orientation on state space methods.

One problem with object orientation is that it is deceptively simple when in fact there is a conglomeration of interrelated concepts. In this talk, we propose to tease out these concepts and separately consider the implications for analysis of modularity, the use of object identifiers, and the relations between classes. We assume the reader has a working knowledge of Coloured Petri Nets (CPNs) and their analysis [15,16].

2 Modularity

One of the key strengths of object orientation is its implementation of abstraction. An object is a module with a clearly defined interface and with a hidden implementation. In other words, an object exhibits the key notions of encapsulation and information hiding, which have long been accepted as fundamental

to the management of complexity [30]. The fact that an object captures an abstraction means that its constituent components belong together, a property identified as *strong cohesion*. On the other hand, the interaction between objects needs to be limited, a property identified as *weak coupling*.

In other words, the external view of an object sees some basic services as specified by the interface. The internal view can focus on the detailed implementation, in the knowledge that there is minimal interaction with the environment. This structuring makes software more intellectually manageable. This is undoubtedly an imprecise notion but it does imply a reduction in the possible interaction between an object and its environment, a property which can lead to a reduced state space given appropriate modular analysis algorithms.

A modular state space analysis algorithm has been proposed for Petri Nets [7]. Here the interfaces between modules are given by transitions which are fused with other interface transitions (of other modules). Thus, the environment of the modules is simply a context for the fusion of various interface transitions. The state space or reachability graph for such a system is built in a modular fashion, with *local reachability graphs* capturing the internal activity of a module, and a *synchronisation graph* capturing the interaction between the modules, i.e. the firing of fused interface transitions [7].

An alternative perspective to the above is to allow the environment of the modules to be a net. This environment net would include transitions corresponding to the fusion of the interface transitions of the modules. It would also include one place for each module, which would hold a single token specifying the entry marking of the module, i.e. the marking following the firing of an interface transition. The marking of such a place would only be affected by interaction with the associated module, in which case the token would be removed and a replacement token added. It would not be affected by global activity or even interaction with other modules. The condition under which such a fused transition can fire is a complex one determined by the internal logic of the module, namely by the local reachability graph. The environment net could (but doesn't have to) include additional net components. With this alternative perspective, the earlier synchronisation graph becomes a *global reachability graph*, and the approach can then be generalised in a straightforward manner to multiple levels of module nesting. Finally, the places associated with modules can hold a token indicating a strongly connected component rather than an individual marking. This reduces the size of global reachability graph (or the earlier synchronisation graph).

By separating the global activity from the local activity of each module, the above modular analysis algorithm avoids elaborating the possible interleavings of global and local activity, which has significant benefits for state space exploration. The extent to which modules exhibit strong cohesion and weak coupling should be the extent to which there are gains to be made by modular analysis.

3 The Use of Object Identifiers

The use of references or object identifiers is fundamental to object orientation. As well as being characterised by having state and behaviour, objects also have a unique identity. This makes it possible for the one object to be referenced and accessed by multiple other objects. Associated with the use of references is the possibility of generating objects dynamically and collecting discarded objects, whether explicitly by use of delete primitives or implicitly via garbage collection.

There are various motivations for supporting the use of references in a formalism. One rationale for the work on Distributed SPIN is the desire to analyse object oriented software [14]. The specification language SDL, widely used in the telecommunications industry, contains object oriented features, including the dynamic generation and discarding of processes [8]. In order to analyse SDL specifications by translating them into some formalism (such as Petri Nets), it is necessary to support some form of references [12]. More generally, the use of references is assumed in the provision of polymorphism and dynamic binding in object oriented languages. Here, a reference to an instance of some class can be replaced by a reference to an instance of a subclass. The context need only assume that the object will provide a suitable implementation for the specified interface. (This raises the question of suitability, to which we return in section 4.) This kind of polymorphism is extensively used in Design Patterns to provide flexible solutions to common problems. Therefore, if one wishes to support this use of polymorphism in some formalism, then references must be included.

It is common to view references or object identifiers as elements of some unbounded data type. The fact that it is unbounded immediately creates serious problems for state space methods. Even if the data type can be bounded (without modifying system behaviour), there are significant problems created by the nondeterministic allocation of references in different concurrent processes. In general, the permutation of object identifiers will lead to similar behaviour and hence some form of symmetry analysis is desirable, if not essential [13]. In some cases, however, the evolution of a particular individual object may be relevant, in which case more detailed information about symmetry reductions will need to be recorded [37]. Sophisticated naming conventions may alleviate the above state space explosion in some cases, but they do not eliminate the need for some form of symmetry analysis [37]. It is common to check for symmetrical states by mapping them into a canonical form and then checking for equality. In dSPIN, the canonical form can be based on a canonical ordering of variables [13], while for Petri Nets it is also necessary to determine a canonical marking for each place [17,25]. For general symmetries, this process can be quite involved [25]; it is somewhat simpler for the arbitrary permutation symmetries of object identifiers.

It is clear from the above that the analysis of formal models which include the use of object identifiers must address the above issues, with the associated processing costs. On the other hand, there are also benefits since the use of references makes it possible to harness polymorphism and the techniques captured by Design Patterns. A simple yet common example is that of being able to conclude that a protocol will successfully transmit protocol data units (PDUs) of some

complex type based on the knowledge that it can successfully transmit PDUs of some more basic supertype. For example, the alternating bit protocol or the sliding window protocol depend only on the header information of a message — the kind of message, the sequence number, acknowledgement number, etc. For state space analysis, it is desirable to eliminate extraneous data [35] and the above approach makes it possible to do so while still being able to infer properties for the more general case.

4 Relations between Classes

Inheritance is the characteristic which distinguishes an object oriented language from one classified as object based [38]. Inheritance is normally associated with *subclassing* (which is concerned with code reuse) as opposed to *subtyping* (which is concerned with substitutability) [2]. We feel that this dichotomy is often exaggerated, since a primary motivation for object orientation has always been code reuse. However, this does not necessarily imply a complete lack of substitutability properties, as is demonstrated by the use of assertions in *Eiffel* [26]. Our particular interest is in the support for incremental development of system specifications, where the designer starts with an abstract view of the system, and progressively refines that specification to add more detail. The very use of abstraction means that the designer has captured some aspects of the system (which will persist henceforth) while consciously deciding to ignore others (at least for the time being). In producing a refined version, existing behaviour is maintained (in some sense), while new behaviour is added. It is appropriate to use inheritance in this context, so that the refined version of a system component is derived from the abstract version.

The designer does not face a stark choice between subclassing and subtyping. Rather, code is being reused and some level of behavioural compatibility is being maintained. Wegner and Zdonik's comments are particularly pertinent here:

> ... the requirements for substitutability and the associated notion of subtype and behavioural compatibility is too strong in many practical situations. ... template modification (which is at the heart of subclassing) is more powerful than subtying as an incremental modification mechanism but also less tractable [38]

Suffice it to say, that our approach has been to sacrifice strong substitutability for the sake of pragmatic applicability [22]. In the context of CPNs, we have proposed three forms of refinement — type refinement, subnet refinement and node refinement [20,21]. *Type refinement* replaces a type in the Petri Net with a subtype, with the constraint that every value of the subtype can be projected on to a corresponding value of the original type. The extension of a protocol data unit with additional data fields would typically qualify as type refinement. *Subnet refinement* augments the Petri Net with additional net components (i.e. nodes and arcs), with the constraint that the additional components do not undermine the original behaviour of the system. The extension of a protocol with

additional functionality would normally qualify as a subnet refinement. Finally, *node refinement* replaces a node in the Petri Net (a place or a transition) by a subnet, with the constraint that a canonical construction is used. Elaborating the details of message processing would normally qualify as node refinement.

All three forms of refinement are constrained so as to maintain behavioural consistency between refined and abstract specifications. This means that every (complete) action sequence of the refined model has a corresponding action sequence in the abstract model. An examination of case studies in the literature has revealed that these forms of refinement with these constraints are applicable in practice, i.e. where case studies are presented in a structured way, the constraints on the proposed forms of refinement do apply [23].

We have used the above behavioural consistency as the basis for specially-tailored incremental analysis algorithms which can build the state space of the refined model from that of the abstract model [24]. In the case of node refinement, the analysis algorithm is based on the modular analysis algorithm described in section 2. Here, the global reachability graph (or synchronisation graph) is essentially the reachability graph for the abstract system except for the addition of entry states for the refined nodes. In some practical cases, particularly for node refinement, the performance improvements for the incremental algorithms are quite dramatic.

As noted above, we have opted for a more general form of node refinement, particularly in the case of transition refinement (which corresponds to the action refinement of [10,36]). It is possible to add further constraints and gain formal properties and/or analysis benefits. For example, if the transition refinement is restricted to a single input and a single output border transition with a live subnet, then we can prove bisimilarity between the abstract and refined systems, and we have the situation which has been extensively analysed by Vogler [36]. Alternatively, if we can guarantee liveness of the output border transitions once all the corresponding input border transitions have occurred, then the analysis can be substantially simplified [23].

5 Conclusions

This talk has sought to explore the benefits for state space exploration of the inclusion of object oriented capabilities into a formalism such as Petri Nets.

Acknowledgements

This work was partially funded under ARC Large Grant A49800926. The author gratefully recognises the helpful discussions and collaboration with the Petri Nets and formal methods research group of the Laboratory for Theoretical Computer Science at the Helsinki University of Technology, Finland, and the Coloured Petri Net group of the Computer Science Department at the University of Aarhus, Denmark.

References

1. G. Agha, F. De Cindio, and G. Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2001. 60
2. P. America. Designing an object-oriented language with behavioural subtyping. In *REX School/Workshop*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90, Noordwijkerhout, The Netherlands, 1995. Springer-Verlag. 59, 63
3. G. Booch. *Object-oriented Analysis and Design with Applications*. Benjamin/Cummings Series in Object-oriented Software Engineering. Benjamin/Cummings, Redwood City, California, 2nd edition, 1994. 59
4. G. Booch, J. Rumbaugh, and I. Jacobsen. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Mass., 1999. 59
5. Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, Reading, Mass, 2000. 60
6. K. Bruce and L. Petersen. Subtyping is not a good "match" for object-oriented languages. In M. Aksit and S. Matsuoka, editors, *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127, Jyväskylä, Finland, 1997. Springer. 59
7. S. Christensen and L. Petrucci. Modular state space analysis of Coloured Petri Nets. In G. De Michelis and M. Diaz, editors, *Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, Berlin, 1995. 61
8. J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-Oriented Language for Communication Systems*. Prentice-Hall, 1996. 62
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison-Wesley, Reading Massachussets, 1994. 59
10. R. Gorrieri and A. Rensink. Action refinement. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1147. Elsevier Science, 2001. 64
11. W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA 93 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 28 of *SIGPLAN Notices*, pages 411–428. ACM, 1993. 60
12. N. Husberg and T. Manner. Emma: Developing an industrial reachability analyser for SDL. In *Formal Methods 1999*, volume 1709 of *Lecture Notes in Computer Science*, pages 642–661, Toulouse, France, 1999. Springer-Verlag. 62
13. R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *16th IEEE Conference on Automated Software Engineeering*, pages 254–261. IEEE, 2001. 62
14. R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In *6th SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 1999. 62
15. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use — Volume 1: Basic Concepts*, volume 26 of *EATCS Monographs in Computer Science*. Springer-Verlag, Berlin, 1992. 60
16. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use — Volume 2: Analysis Methods*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1994. 60

17. T. A. Junttila. Finding symmetries of Algebraic System Nets. *Fundamenta Informatica*, 37:269–289, 1999. 62
18. Wolfram Kaiser. Become a programming Picasso with JHotDraw: Use the highly customizable GUI framework to simplify draw application development. *JavaWorld*, February 2001. 59
19. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Finland, 1997. Springer-Verlag. 59
20. C. A. Lakos. On the abstraction of Coloured Petri Nets. In P. Azéma and G. Balbo, editors, *18th International Conference on the Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 42–61, Toulouse, France, 1997. Springer-Verlag. 63
21. C.A. Lakos. Composing abstractions of Coloured Petri Nets. In M. Nielsen and D. Simpson, editors, *International Conference on the Application and Theory of Petri Nets*, volume 1825 of *Lecture Notes in Computer Science*, pages 323–342, Aarhus, Denmark, 2000. Springer. 63
22. C.A. Lakos and G.A. Lewis. A practical approach to incremental specification. In S. Smith and C. Talcott, editors, *Fourth International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 233–256, Stanford, California, 2000. Kluwer. 63
23. G.A. Lewis. *Incremental Specification and Analysis in the Context of Coloured Petri Nets*. Phd, Department of Computing, University of Tasmania, 2002. 64
24. G.A. Lewis and C.A. Lakos. Incremental state space construction of Coloured Petri Nets. In *International Conference on the Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, Newcastle, U.K., 2001. Springer. 64
25. L. Lorentsen and L. Kristensen. Exploiting stabilizers and parallelism in state space generation with the symmetry method. In *2nd International Conference on Application of Concurrency to System Design*, pages 211–220, Newcastle, U.K., 2001. IEEE Computer Society. 62
26. B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992. 59, 63
27. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 2nd edition, 1997. 59
28. O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992. 59
29. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. Wiley Professional Computing. Wiley, Chichester, 1994. 59
30. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972. 61
31. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, 1991. 59
32. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, 1996. 60
33. J. Siegel. *CORBA Fundamentals and Programming*. Wiley, New York, 1996. 59
34. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998. 60
35. A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, Dagstuhl, 1998. 60, 63

36. W. Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992. 64
37. T. Vojnar. *Towards Formal Analysis and Verification over State Spaces of Object-Oriented Petri Nets*. PhD thesis, Department of Computer Science and Engineering, Brno University of Technology, 2001. 62
38. P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism, or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *ECOOP '88 - European Conference on Object Oriented Programming*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77, Oslo, Norway, 1988. Springer Verlag. 63
39. A. Weinand, E. Gamma, and R. Marty. ET++ – An object-oriented application framework in C++. In *OOPSLA 88 Conference on Object Oriented Programming, Systems, Languages and Applications*. ACM, 1988. 59
40. D. Wong, N. Paciorek, and D. Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):92–102, 1999. 59

Abstract Cyclic Communicating Processes: A Logical View

P. S. Thiagarajan*

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

Abstract. Temporal logics have been very successful as tools for specifying and verifying dynamic properties of finite state distributed systems. An intriguing fact is that these logics, in order to be effective, must semantically filter out at least one of the two basic features of the behavior of distributed systems: indeterminacy and concurrency. To be precise, linear time temporal logics use as models the interleaved runs of a system in which both indeterminacy (i.e. the choices presented to the system) and concurrency (i.e. causally independent occurrences of actions) have been, in some sense, defined away. In branching time temporal logics, the (interleaved) runs glued together into a single object -usually called a computation tree- serves as a model. Here, speaking again loosely, indeterminacy is present through the branching nature of the computation tree but information regarding concurrency has been filtered out. There are also families of linear time temporal logics which are interpreted over the partially ordered runs (often represented as Mazurkiewicz traces) of a system. In this setting, concurrency is in but indeterminacy is out.

It seems difficult to come up with *effective* temporal logics in which both indeterminacy and concurrency are explicitly handled. More precisely, the natural logics that one comes up with turn out to be undecidable in terms of the associated satisfiability problems. To bring this out, let $\mathcal{N} = (B, E, F, M_{in})$ be a finite 1-safe Petri net. It is well known that the non-interleaved branching time behavior of $\mathcal{N}$ can be represented as a labeled prime event structure $ES_{\mathcal{N}} = (\widehat{E}, \leq, \#, \lambda)$. Here $\lambda : \widehat{E} \rightarrow E$ is the labeling function. Let $MSO(\mathcal{N})$ be the monadic second order logic (similar in spirit to S2S) interpreted over $ES_{\mathcal{N}}$ and whose syntax is given by:

$MSO(\mathcal{N}) ::= Q_b(x) \mid x \leq y \mid x \# y \mid x \in X \mid \sim \varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x(\varphi) \mid \exists X(\varphi)$

Here b ranges over B, the set of S-elements of the 1-safe Petri net $\mathcal{N}$. As for the semantics, it will be convenient to augment $\widehat{E}$ with a new element $\perp$. With each finite configuration c of $ES_{\mathcal{N}}$ one can associate in a natural way a unique reachable marking of $\mathcal{N}$ denoted $Mark(c)$. Now the individual variables are interpreted as members of $\widehat{E} \cup \{\perp\}$ and the set variables are interpreted as subsets of $\widehat{E} \cup \{\perp\}$. Under the interpretation $\mathcal{I}$, $Q_b(x)$ is true iff $b \in Mark(\downarrow \mathcal{I}(x))$ where,as usual, $\downarrow \mathcal{I}(x)$ is the finite configuration $\{e \mid e \leq \mathcal{I}(x)\}$. We set $\downarrow(\perp) = \emptyset$.

* On leave from Chennai Mathematical Institute, Chennai, India

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 68–69, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The rest of the semantics is along expected lines. It should be clear that in this logic we can define and compare configurations. Hence quite a lot can be said about the computation paths through the configuration space.

The sentence φ (i.e. it has no free individual or set variables) is satisfiable iff $ES_{\mathcal{N}} \models \varphi$. The satisfiability problem for $\mathcal{N}$ is to determine whether or not a sentence in $MSO(\mathcal{N})$ is satisfiable.

It is easy to come up with 1-safe Petri nets for which the satisfiability problem is undecidable. Here is an example:

Let $\mathcal{N}_{grid} = (\{X, Y\}, \{r, u, col\}, F, M_{in})$ where
$F = \{(X, r), (r, X), (Y, u), (u, Y), (X, col), (Y, col)\}$ and $M_{in} = \{X, Y\}$. It is easy to show that the satisfiability problem for $\mathcal{N}_{grid}$ is undecidable.

It follows from known results that if the underlying net of $\mathcal{N}$ is an S-graph or a T-graph then the satisfiability problem for $\mathcal{N}$ is decidable.

Our goal is to exhibit a new class of 1-safe Petri nets for which the satisfiability problem is decidable although their underlying nets may be quite complex. This class arises by blowing up abstract net systems that correspond to 1-safe colored marked graphs but where the color domains are finite.. The latter objects can be naturally viewed as *cyclic* processes that manipulate finite data values and communicate by synchronization actions. Such a network of processes often correspond to *open* systems whose communication patterns are deterministic. A typical example is a ring network where each node interacts with its environment, does an internal computation, talks to its left neighbor, talks to its right neighbor; and this pattern is repeated endlessly.

A simple but useful rephrasing of our main result is that the monadic second order theory of abstract cyclic communicating processes (in the sense described above) is decidable.

Is the Die Cast for the Token Game?

Alex Yakovlev

Department of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, U.K.
Alex.Yakovlev@ncl.ac.uk

Abstract. As semiconductor technology strides towards billions of transistors on a single die, problems concerned with deep submicron process features and design productivity call for new approaches in the area of behavioural models. This lecture focuses on some of recent developments and new opportunities for Petri nets in designing asynchronous circuits such as synthesis of a good net-based 'backend' in the asynchronous design flow and performance-oriented mapping of nets to circuits.

1 Messages from Semiconductor Technology Roadmap

The International Technology Roadmap for Semiconductors predicts the end of this decade will be marked by the appearance of a System-on-a-Chip (SoC) containing four billion 50-nm transistors that will run at 10GHz. With a steady growth of about 60% in the number of transistors per chip per year, following the famous Moore's law, the functionality of a chip doubles every 1.5 to 2 years. Such a SoC will inevitably consist of many separately timed communicating domains, regardless of whether they are internally clocked or not [1]. Built at the deep submicron level, where the effective impact of interconnects on performance, power and reliability will continue to increase, such systems present a formidable challenge for design and test methods and tools.

The key point raised in the Roadmap is that *design cost* is the greatest threat to the continued phenomenal progress in microelectronics. The only way to overcome this threat is through improving the productivity and efficiency of the design process, particularly by means of design automation and component reuse. The cost of design and verification of processing engines has reached the point where thousands of man-years are spent to a single design, yet processors reach the market with hundreds of bugs [1].

2 Self-Timed Systems and Design Tools

Getting rid of global clocking in SoCs offers potential added values, traditionally quoted in the literature [2]: greater operational robustness, power savings, electro-magnetic compatibility and self-checking. While the asynchronous design community continues its battle for the demonstration of these features to the semiconductor industry investors, the issue of design productivity may suddenly turn the die to the right side for asynchronous design. Why?

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 70–79, 2002.
© Springer-Verlag Berlin Heidelberg 2002

One of the important sub-problems of the productivity and reuse problem for globally clocked systems is that of timing closure. This issue arises when the overall SoC is assembled from existing parts, called Intellectual Property (IP) cores, where each part has been designed separately (perhaps even by a different manufacturer) for a certain clock period, assuming that the clock signal is delivered accurately, at the same time, to all parts of the system. Finding the common clocking mode for SoCs that are built from multiple IP cores is a very difficult problem to resolve.

Self-timed systems, or less radical, globally asynchronous locally synchronous (GALS) systems, are increasingly seen by industry as a natural way of composing systems from predesigned components without the necessity to solve the timing closure problem in its full complexity. As a consequence, self-timed systems highlight a promising route to solving the productivity problem as companies begin to realise. But they also begin to realise that without investing into design and verification tools for asynchronous design the above promise will not materialise. For example, Philips, whose products are critical to the time-to-market demands, is now the world leader in the exploitation of asynchronous design principles [3]. Other microelectronics giants such as Intel, Sun, IBM and Infineon, follow the trend and gradually allow some of their new products involve asynchronous parts. A smaller 'market niche' company Theseus Logic appears to be successful in down-streaming the results of their recent investment in asynchronous design methods (Null-Convention Logic) [4].

3 Design Flow Problem

The major obstacle now is the absence of a flexible and efficient design flow, which must be compatible with commercial CAD tools, such as for example the Cadence toolkit. A large part of such a design flow would be typically concerned with mapping the logic circuit (or sometimes macro-cell) netlist onto silicon area using place and route tools. Although hugely important this part is outside our present scope of interest. What we are concerned with is the stage in which the behavioural specification of a circuit is converted into the logic netlist implementation.

The pragmatic approach to this stage suggests that the specification should appear in the form of a high-level Hardware Description Language (HDL). Examples of such languages are the widely known VHDL and Verilog, as well as Tangram or Balsa that are more specific for asynchronous design. The latter are based on the concepts of processes, channels and variables, similar to Hoare's CSP.

We can in principle be motivated by the success of behavioural synthesis achieved by synchronous design in the 90s. However, for synchronous design the task of translating an HDL specification to logic is fairly different from what we may expect in the asynchronous case.

Its first part was concerned with the so-called architectural synthesis, whose goal was the construction of a register-transfer level (RTL) descrtiption. This

required extracting a control and data flow graph (CDFG) from the HDL, and performing scheduling and allocation of data operations to functional datapath units in order to produce an FSM for a controller or sequencer. The FSM was then constructed using standard synchronous FSM synthesis, which generated combinational logic and rows of latches.

Although some parts of architectural synthesis, such as CDFG extraction, scheduling and allocation, might stay unchanged for self-timed circuits, the development of the intermediate level, an RTL model of a sequencer, and its subsequent circuit implementation, would be quite different.

4 How Can Petri Net Help?

Two critical questions arise at this point. Firstly, what is the most adequate formal language for the intermediate (still behavioural) level description? Secondly, what should be the procedure for deriving logic implementation from such a description?

The present level of development of asynchronous design flow sugggests the following options to answer those questions:

(1) Avoid (!) answering them altogether. Instead, follow a syntax-driven translation of the HDL directly into a netlist of hardware components, called handshake circuits. This sort of silicon-compilation approach was pursued at Philips with the Tangram flow [5]. Many computationally hard problems involving global optimisation of logic were also avoided. Some local 'peephole' optimisation was introduced at the level of handshake circuit description. Petri nets were used for that in the form of Signal Transition Graphs (STGs) and their composition, with subsequent synthesis using the Petrify tool [7,9]. Similar sort of approach is currently followed by the designers of the Balsa flow, where the role of peephole optimisation tools is played by the FSM-based synthesis tool Minimalist [11]. The problem with this approach is that, while being very attractive from the productivity point of view, it suffers from the lack of global optimisation, especially for high-speed requirements, because direct mapping of the parsing tree into a circuit structure may produce very slow control circuits.
(2) Translate the HDL specification into an STG for controller part and then synthesise this STG using Petrify. This approach was employed in [6], where the HDL was Verilog. This option was attractive because the translation of the Verilog constructs preserved the natural semantical execution order between operations (not the syntax structure!) and Petrify could apply logic optimisation at a fairly global level. If the logic synthesis stage was not constrained by the state space explosion inherent in Petrify, this would have been an ideal situation.

However, the state space size becomes a real spanner in the works, because the capability of Petrify to solve the logic synthesis problem is limited by the

number of logic signals in the specification. STGs involving 40-50 binary variables can take hours of CPU time. The size of the model is critical not only for logic minimisation but, more importantly, for solving state assignment and logic decomposition problems. The state assignment problem often arises when the STG specification is extracted automatically from an HDL. This forces Petrify into solving Complete State Coding (CSC) using computationally intensive procedures involving calculation of regions in the reachability graph.

While the logic synthesis powers of Petrify should not be underestimated, one should be realistic *where* they can be applied efficiently. Thus the solution lies where the design productivity similar to that of (1) can be achieved together with the circuit optimality offered by (2).

The proposed design flow is shown in Figures 1 and 2. Firstly, it involves the construction of a labelled Petri net (LPN) model of top-level control from the HDL specification together with the low-level control described by Signal Transition Graphs (STG). Secondly, it performs transformation of the LPN model to the form which guarantees high-performance circuit implementation. Thirdly, the new LPN model is directly mapped to the implementation netlist using a set of mapping rules and a library of cells associated with the constructs in the LPN. For the low-level control, STGs can be synthesised either using Petrify or the same direct mapping procedures.

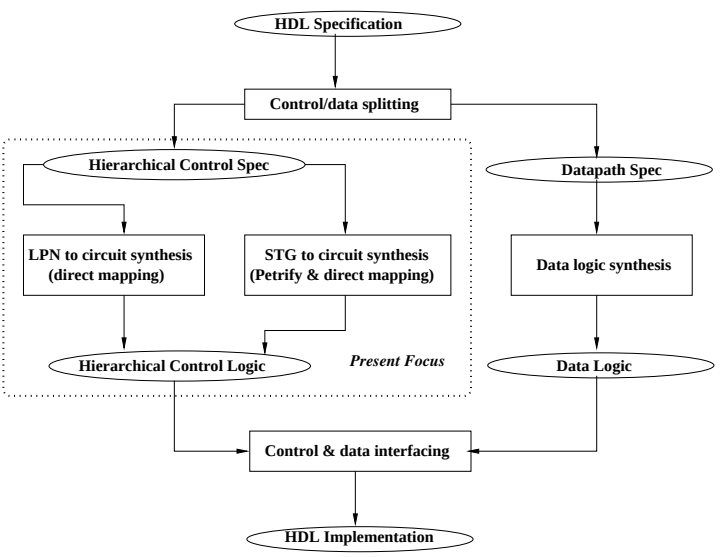


Fig. 1. Proposed Design Flow

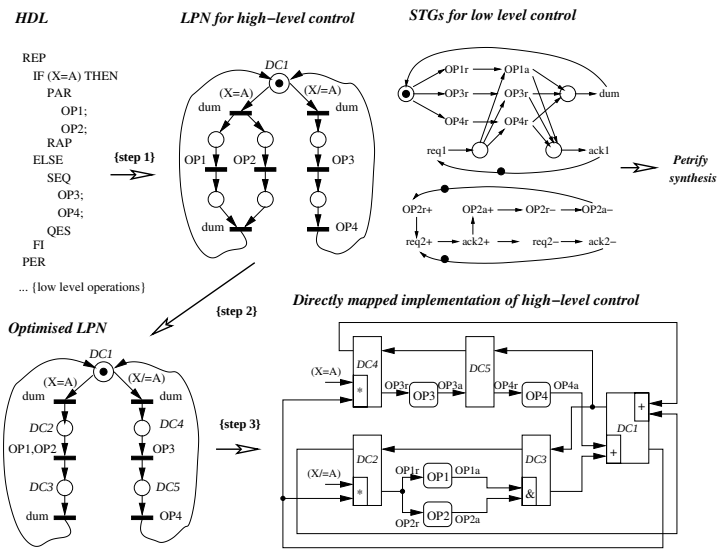


Fig. 2. Two-level control synthesis

5 Bridging the Gap between Petri Nets and Circuits

The quality and efficiency of the new design flow will be determined by the availability of efficient methods and algorithms for: (a) translation of HDLs into Petri nets, (b) manipulation of Petri nets under the criteria dictated by hardware synthesis, and (c) direct mapping of Petri nets to circuits.

The development of these techniques depends on the following theoretical issues:

(a) Rigorous semantic relationship between control-flow constructs used in typical behavioural HDLs and their equivalents in LPNs. For example, if one uses Balsa, such constructs basically include sequencing, parallelisation, two-way and multi-way selection and (forever, while and for) loops, as well as macro and procedure calls. Those can be translated into Petri nets quite efficiently as done for example in PEP[8] for the translation of basic high-level programming language notation(B(PN)^2 into M-nets. However, the HDL-to-LPN translation in PEP was aimed at subsequent verification and model checking, which did not put specific constraints on the structure of the resulting net. In our case, such constraints would be imposed because the net would ultimately be translated to a circuit that retains the same structure.

(b) Semantic preserving transformation of LPNs from its initial form, called a specification net (SN), produced by the HDL-to-LPN translator to the form which is amenable to direct mapping to a circuit. The latter is called an implementation net (IN). The transformations involved here are various forms of refinement and contraction, involving insertion and removal of dummy

events and changing concurrency and choice structure to a certain extent. It is important that the key elements of the ordering relation between some transitions or their labels are preserved in the IN. However, in order to make significant speed optimisations, these relations have to be suficiently flexible (for example, they may be constrained by the datapath conflict conditions obtained from the HDL description). This stage needs to be supported by tools for analysis of performance (worst or average case cycle times, fanins and fanouts of places and transitions) and possibly other criteria (such as power consumption or silicon area utilisation).

(c) Rigorous semantic relationship between elements of the IN and circuit components. A particular emphasis is made on the structure of the net and its circuit implementation. This structure should clearly distinguish parts of the net model capturing the interface between the circuit and the environment, and parts related to the internal control behaviour.

In this lecture we focus our attention on the last issue, which provides us with the definition of the IN. Informally, our goal is to establish the closest possible semantic link between control circuits and Petri nets from the point of view of the direct mapping. This link should also support efficient composition of circuits in terms of nets. The key feature in this composition is synchronisation by means of *read arcs*. In the context of direct mapping, the latter closely corresponds to the interconnection of circuit components by means of wires.

6 Synthesis via Direct Mapping

The idea of direct mapping of a Petri net to hardware is not new. In various forms it appeared in [12,13,14,15,16]. In the framework of 1-safe Petri nets and speed-independent cirsuits this problem was solved in [15], however only for autonomous (no inputs) specification where all operations were initiated by the control logic specified by an LPN. Another limitation was that the technique did not cover nets with arbitrary dynamic conflicts. Hollaar's one-hot encoding method [14] allowed explicit interfacing with the environment but required fundamental mode timing conditions, use of internal state variables as outputs and could not deal with conflicts and arbitration in the specifications. Patil's method [12] works for the whole class of 1-safe nets. However, it produces control circuits whose operation uses 2-phase (non-return-to-zero) signalling. This results in lower performance than what can be achieved for 4-phase circuits used in [15].

The direct mapping technique presented in this lecture produces 4-phase logic. It is capable of translating an arbitrary 1-safe LPN specification with explicit input/output interface into an equivalent self-timed circuit, and guarantees minimum latency between inputs and outputs.

Figure 3 illustrates the transformation of a specification net to the form that can be directly mapped into a logic circuit. Part (a) of the figure depicts the main idea of separation of the system net into circuit and environment

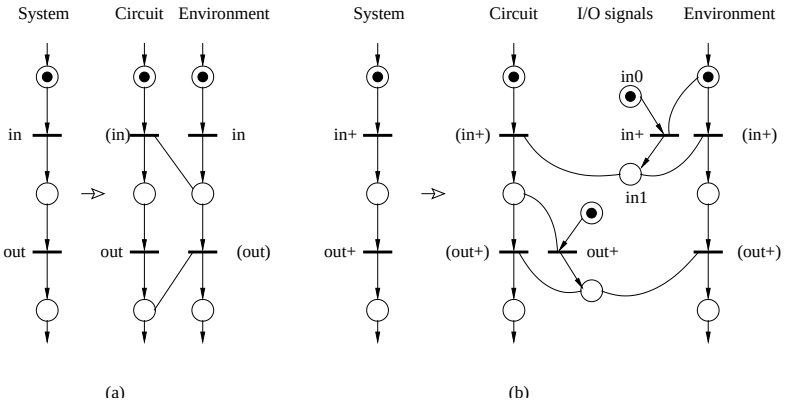


Fig. 3. The idea of tracking and input/output exposition

subnets and their synchronisation by means of read arcs. Part (b) applies this idea to the case where the system model is an STG. Here, the original STG is decomposed into the circuit subnet, called the (state) tracker, environment subnet and subnets modelling input and output (binary) signals. Each of the I/O subnets is defined as a pair of complementary places, for the values of 0 and 1 respectively, and with up-going and down-going transitions of the signal. The synchronisation between these parts is done entirely by read arcs. In the course of the transformation some transitions (their labels are shown in parentheses) are made internal (dummy). The transformation produces a model whose behaviour is equivalent (weak bisimulation) to the original STG with respect to the original set of signal labels.

To derive the circuit implementation we only use the tracker and I/O subnets. Figure 4 presents a simple example in which a control logic is synthesised for a weakly coupled pair of handshakes (a, x) and (b, y), where a and b are inputs and x and y are outputs. The original STG is subject to the decomposition of the system net into the tracker and I/O subnets. After that an optimisation is applied to the tracker net, in which some internal events together with their places are deleted. This helps reduce the size and delay of the tracker logic that will be obtained by direct mapping from the tracker net. The Petri nets are mapped into circuits as follows. Each place in the tracker net is associated with a simple memory cell, called a David cell, whose internal logic and connectivity with predecessor and successor cells is determined by the type of connections of the place in the net. The connections between adjacent David cells and their set and reset logic are associated with the transitions in the tracker net and their consuming and producing arcs. The output signal fragments are implemented by flip-flops whose set and reset signals are connected with the appropriate signals in the tracker logic according to how read arcs connect set and reset (e.g. $x+$ and $x-$) transitions with the state-holding places in the tracker net (e.g. $p1$ to $x-$ and $p4$ to $x+$). Inputs are connected, in their direct, a, or inverted form, $\overline{a}$,

to the set conditions of the appropriate David cells according to how the values of input signals (e.g., $a0$ and $a1$) precondition the corresponding transitions in the tracker net.

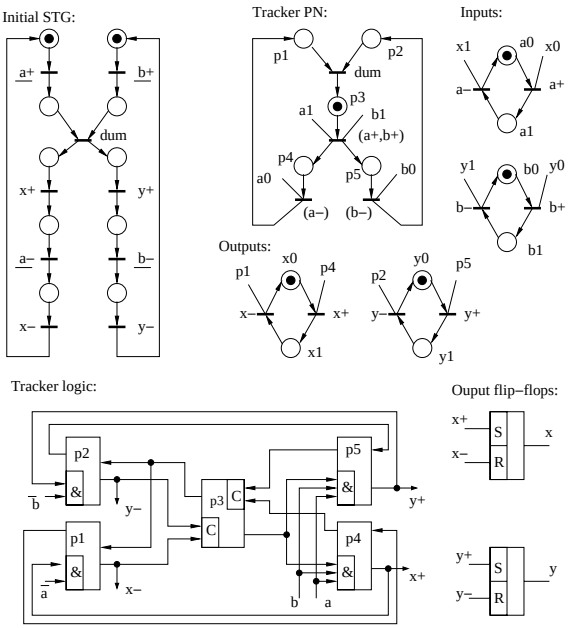


Fig. 4. Direct mapping of STG: decoupled handshake example (inputs: a and b, outputs: x and y)

Subsequent size and speed optimisation can be achieved by rearranging the conditions for switching the outputs so that the latency between an input transition and the subsequent ouput tranistion is minimal. This requires connecting transitions in the output nets by read arcs directly to the places in the input nets. For example, the condition $p4$ (from the tracker) to fire $x+$ can be replaced by a pair of conditions $p3$ (from the tracker) and $a1$ (from the signal a net). Other local transfromations for speed are possible such as use of non-speed-independent implementations of David cells, in which relative timing assumptions are used.

The details of this direct mapping technique together with the speed optimisations can be found in [17,18]. These papers demonstrate that while the direct mapping approach has no alternative in designing large scale controllers from LPNs it is also quite competitive with logic synthesis by Petrify for relatively small scale circuits. The use of direct mapping in practical designs has been presented in [19,20].

In the lecture we will also look at ways of direct mapping of Petri nets with dynamic conflicts, such as multi-way arbiters with confusion.

7 Further Research

A number of problems in the use of Petri nets for asynchronous circuit design urgently need practically applicable solutions – here are some of them:

- Compilation of HDLs to LPNs that can be efficiently implemented by direct mapping. In particular, efficient partitioning of HDL constructs into data and control parts is important. Analysis and comparison of solutions based on separate data and control, where the latter is captured by low-level PNs, with those where data and control are combined through the use of high-level nets is of particular interest.
- Transformation and structural optimisation of PNs with respect to various cost functions, such as cycle time, response time (between inputs and outputs). In such transformations the positions of dummies will matter!
- Testing and testability LPNs for various faults: stuck-at transitions (that are enabled infinitely long without firing), stuck-at token in a place, disappearance of a token from a place, "cross-talk" (token switches from one place to another), etc.

Acknowledgements

In the early 80s, Victor Varshavsky introduced me into the art of constructing self-timed circuits using direct translation of Petri nets and parallel program schemata. Recently, I pursued this approach with my Newcastle colleagues: Alex Bystrov, Frank Burns, Delong Shang, Danil Sokolov and Fei Xia. EPSRC is supporting this work through GR/R16754.

References

1. A. Allan, et. al., 2001 Technology Roadmap for Semiconductors, *Computer*, January 2002, pp. 42-53. 70
2. J. Sparsø and S. Furber, Eds., Principles of Asynchronous Circuit Design: A Systems Perspective. Kluwer Academic Publishers, 2001. 70
3. S. Furber, Industrial take-up of asynchronous design, Keynote talk at the Second ACiD-WG Workshop, Munich, Jan. 2002 (http://www.scism.sbu.ac.uk/ccsv/ACiD-WG/Workshop2FP5/Programme/). 71
4. D. Ferguson and M. Hagedorn, The Application of NULL Convention Logic to Microcontroller/Microconverter Product, Second ACiD-WG Workshop, Munich, Jan. 2002 (http://www.scism.sbu.ac.uk/ccsv/ACiD-WG/Workshop2FP5/Programme/). 71
5. K. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of International Series on Parallel Computation. Cambridge University Press, 1993. 72
6. I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL, *Proc. of IEEE Symp. on Adv. Res. in Async. Cir. and Syst. (ASYNC 2000)*, IEEE CS Press, pp. 84–92. 72

7. M. A. Peña and J. Cortadella, Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits, *Proc. of IEEE Symp. on Adv. Res. in Async. Cir. and Syst. (ASYNC'96)*, IEEE CS Press, pp. 222-232. 72
8. E. Best and B. Grahlmann. PEP – more than a Petri Net Tool. *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, Springer-Verlag, Lecture Notes in Computer Science 1055, Springer-Verlag (1996) 397-401. 74
9. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*, Springer-Verlag, 2002, ISBN 3-540-43152-7. 72
10. Petrify, Version 4.0. (see http://www.lsi.upc.es/~jordic/petrify/petrify-4.0/).
11. T. Chelcea, A. Bardsley, D. Edwards and S. M. Nowick. A burst-mode oriented back-end for the Balsa synthesis system, *Proc. of Design, Automation and Test in Europe (DATE'02)*, IEEE CS Press, pp. 330-337. 72
12. S. S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCON*, pages 223–226, 1972. 75
13. R. David. Modular design of asynchronous circuits defined by graphs. *IEEE Transactions on Computers*, 26(8):727–737, August 1977. 75
14. L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982. 75
15. V. I. Varshavsky and V. B. Marakhovsky. Asynchronous control device design by net model behavior simulation. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996*, volume 1091 of *Lecture Notes in Computer Science*, pages 497–515. Springer-Verlag, June 1996. 75
16. A. Yakovlev and A. Koelmans. Petri nets and Digital Hardware Design *Lectures on Petri Nets II: Applications. Advances in Petri Nets, Lecture Notes in Computer Science*, vol. 1492, Springer-Verlag, 1998, pp. 154-236. 75
17. A. Bystrov and A. Yakovlev. Asynchronous Circuit Synthesis by Direct Mapping: Interfacing to Environment, *Proc. ASYNC'02*, Manchester, April 2002. 77
18. D. Shang, F. Xia and A. Yakovlev. Asynchronous Circuit Synthesis via Direct Translation, *Proc. Int. Symp. on Cir. and Syst. (ISCAS'02)*, Scottsdale, Arizona, May 2002. 77
19. A. Yakovlev, V. Varshavsky, V. Marakhovsky and A. Semenov. Designing an asynchronous pipeline token ring interface, *Proc. of 2nd Working Conference on Asynchronous Design Methdologies, London, May 1995*, IEEE Comp. Society Press, N. Y., 1995, pp. 32-41. 77
20. A. Yakovlev, S. Furber and R. Krenz, Design, Analysis and Implementation of a Self-timed Duplex Communication System, CS-TR-761, Dept. Computing Science, Univ. of Newcastle upon Tyne, March 2002. (http://www.cs.ncl.ac.uk/people/alex.yakovlev/home.informal/some_papers/duplex-TR.ps) 77

Quasi-Static Scheduling of Independent Tasks for Reactive Systems

Jordi Cortadella[1], Alex Kondratyev[2], Luciano Lavagno[3], Claudio Passerone[3], and Yosinori Watanabe[2]

[1] Universitat Politècnica de Catalunya
Barcelona, Spain
jordi@lsi.upc.es
[2] Cadence Berkeley Labs
Berkeley, USA
{kalex,watanabe}@cadence.com
[3] Politecnico di Torino, Italy
{lavagno,claudio.passerone}@polito.it

Abstract. The synthesis of a reactive system generates a set of concurrent tasks coordinated by an operating system. This paper presents a synthesis approach for reactive systems that aims at minimizing the overhead introduced by the operating system and the interaction among the concurrent tasks. A formal model based on Petri nets is used to synthesize the tasks. A practical application is illustrated by means of a real-life industrial example.

1 Introduction

1.1 Embedded Systems

Concurrent specifications, such as dataflow networks [9], Kahn process networks [7], Communicating Sequential Processes [6], synchronous languages [4], and graphical state machines [5], are interesting because they expose the inherent parallelism in the application. However, their mixed hardware-software implementation on heterogeneous architectures requires to solve a fundamental *scheduling problem*. We assume in the following that the preliminary *allocation* problem of functional processes to architectural resources has been solved, either by hand or by some appropriate heuristic algorithm. The task of this paper is to define and solve the scheduling problem for a portion of a functional specification allocated to a single processor.

Most embedded systems are *reactive* in nature, meaning that they must process inputs from the environment at the speed and with the delay dictated *by the environment*. Scheduling of reactive systems thus is subject to two often contradicting goals: (1) satisfying timing constraints and (2) using the computing power without leaving the CPU idle for too long.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 80–100, 2002.
© Springer-Verlag Berlin Heidelberg 2002

1.2 Static and Quasi-Static Scheduling

Static scheduling techniques do most of the work at compile-time, and are thus suitable for safety-critical applications, since the resulting software behavior is highly predictable [8] and the overhead due to task context switching is minimized. They may also achieve very high CPU utilization if the rate of arrival of inputs to be processed from the environment has *predictable regular rates* that are reasonably known at compile time.

Static scheduling, however, is limited to specifications without choice (Marked Graphs or Static Dataflow [9]). Researchers have recently started looking into ways of computing a static execution order for operations as much as possible, while leaving data-dependent choices at run-time. This body of work is known as *Quasi-Static Scheduling* (QSS) [1,10,12,2,13]. The QSS problem, i.e. the existence of a sequential order of execution that ensures no buffer overflow, has been proven to be undecidable by [1] for specifications with data-dependent choices . Our work fits in the framework proposed by [12,2], in which Petri nets (PNs) are used as an abstract model, that hides away correlations among choices due to the value of data that are being passed around, and thus achieves a two-fold improvement over [1]:

1. undecidability has not been proven nor disproven. It remains an interesting open problem except in the decidable cases of Marked Graphs [9] and Free-Choice Petri nets [12].
2. powerful heuristics, based on the theory discussed in this paper, can be used to speed up the identification of a solution, if it exists.

In the rest of the paper, we define various scheduling problems for Petri nets, and motivate their practical interest by showing how concurrent programs communicating via FIFO queues can be implemented as software tasks running under a Real Time Operating System. In particular, we use a game-theoretic formulation, in which the scheduler must win against an adversary who can determine the outcome of non-deterministic choices, by avoiding overflow of FIFO queues. The scheduler can resolve concurrency in an arbitrary fashion, but is not allowed to "starve" any input by indefinitely refusing to service it.

1.3 Specification Model

We consider a system to be specified as a set of concurrent processes, similar to those discussed in [3]. A set of input and output ports are defined for each process, and point-to-point communication between processes occurs through uni-directional FIFO queues between ports. Multi-rate communication is supported, i.e. the number of objects read or written by a process at any given time may be an arbitrary constant.

Each communication action on a port, and each internal computation action is modeled by a *transition* in a corresponding Petri net, while places are used to represent both sequencing within processes and FIFO communication.

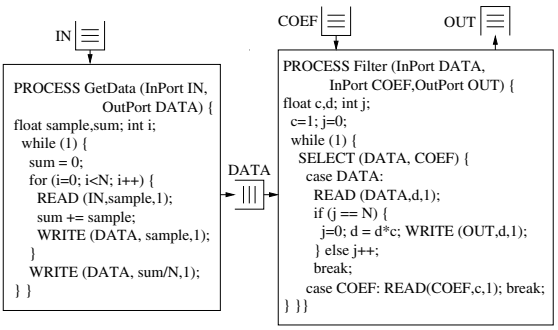


Fig. 1. System specification

Fig. 1 depicts the specification of a concurrent system with two processes, two input ports (`IN` and `COEF`) and one output port (`OUT`). The processes communicate to each other through the channel `DATA`. The process `Get_Data` reads data from the environment and sends it to the channel `DATA`. Moreover, after having sent N samples (N is a constant), it also inserts their average value in the same channel. The process `Filter` extracts the average values inserted by `Get_Data`, multiplies them by a coefficient and sends them to the environment through the port `OUT`. This example also illustrates the main extensions of the C language to support communication. The operations to communicate through ports have syntax `READ_DATA (port, data, nitems)` and `WRITE_DATA (port, data, nitems)`. The parameter `nitems` indicates the number of objects involved in the communication. This allows to support multi-rating, although the example uses only 1-object read/write operations. A `READ_DATA` blocks when the number of items in the channel is smaller than `nitems`.

The `SELECT` statement supports synchronization-dependent control, which specifies control depending on the availability of objects on input ports. In the example, the `SELECT` statement in `Filter` non-deterministically selects one of the ports with available objects. In case none of them has available objects, the process blocks until some is available. Figure 2(a) depicts the representation of the concurrent system specified in Fig. 1 with a Petri net model.

1.4 Schedule Composition

The main topic of this paper is *schedule composition*. In general a schedule is defined by requiring the system to alternately wait for any one input transition, and perform some internal computation or communication. While this is a very useful theoretical notion, in practice it is much easier to tie inputs of the embedded software to interrupt sources of the processor on which it runs.

Basically, if we were able to define fragments of the schedule so that each of them represented the "maximal amount of work" that could be performed in reaction to each input from the environment, then each such fragment would become an interrupt service routine. When defining such schedule fragments we

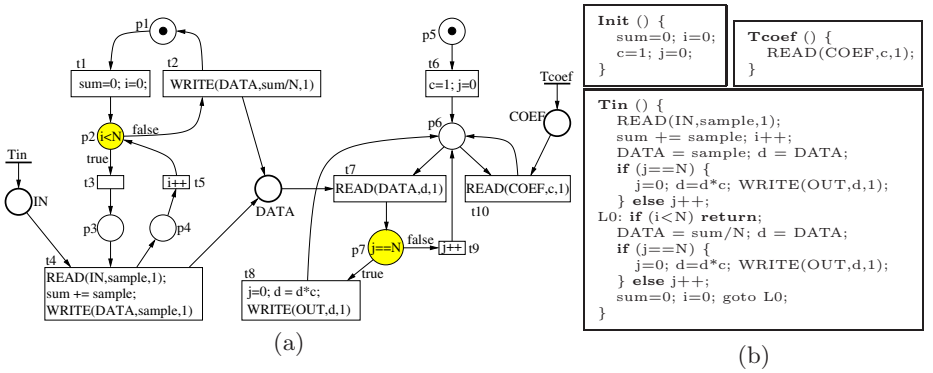


Fig. 2. (a) Petri net specification, (b) Single-source schedules

will mostly focus on their execution in a non-preemptive way, i.e. so that only one of them can be executed at any given time. This results in a simple and general definition of composite schedule, and has the only practical problem that some buffering may be needed on fast inputs in order to process and store the input data while another schedule fragment (called Single Source Schedule in the following) for another input is being executed.

As an example, Fig. 2(b) shows three tasks that implement the behavior of Fig. 1. The task `Init` is executed only at the beginning to reach a steady state. After that, the tasks `Tin` and `Tcoef` are invoked by the arrival of events from `IN` and `COEF`, respectively.

2 Background

The following definitions introduce the nomenclature used in the paper.

Definition 1 (Petri net). *A Petri net is a 4-tuple $N = (P, T, F, M_0)$, where P is the set of places, T is the set of transitions, $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow relation and $M_0 : P \to \mathbb{N}$ is the initial marking. The set of reachable markings of a Petri net is denoted by $[M_0\rangle$. The fact that M' is reachable from M by firing transition t is denoted by $M[t\rangle M'$. The pre-set and post-set of a node $x \in P \cup T$ are denoted by ${}^\bullet x$ and $x^\bullet$, respectively.*

Given a Petri net N with $P = (p_1, \ldots, p_n)$, the notation $\mathsf{Pre}[t]$ is used to represent the vector $(F(p_1, t), \ldots, F(p_n, t))$. Given a set of nodes X, $N \setminus \{X\}$ denotes the subnet of N obtained by removing the nodes in X and their adjacent arcs from N. If for any node x in PN N we have ${}^\bullet x \cap x^\bullet = \emptyset$, then N is called self-loop free. $M(p)$ denotes a number of tokens in place p under marking M.

In this paper we use nets with *source* transitions, i.e. with empty pre-sets. These transitions model the behavior of the input stimuli to a reactive system.

Definition 2 (Source and non-source transitions). *The set of transitions of a Petri net is partitioned into two subsets as follows:*

$$T_S = \{t \in T \mid {}^\bullet t = \emptyset\}, \;\; T_N = T \setminus T_S.$$

T_S and T_N are the sets of source *and* non-source *transitions, respectively.*

Definition 3 (Free-choice set). *A* Free-choice Set *(FCS) is a maximal subset of transitions C such that*

$$\forall t_1, t_2 \in C \text{ s.t. } t_1 \neq t_2 : \;\; \mathit{Pre}[t_1] = \mathit{Pre}[t_2] \;\wedge\; ({}^\bullet t_1 \neq \emptyset \implies C = ({}^\bullet t_1)^\bullet).$$

Proposition 1. *The set of FCSs is a partition of the set of transitions.*

Proof. The proof immediately follows from the consideration that the relation R induced by FCS (i.e. $t_1 R t_2 \iff \exists \mathsf{FCS}\ C : t_1, t_2 \in C$) is an equivalence relation.

We will call $\mathsf{FCS}(t)$ the set of transitions that belong to the same FCS of t. Any conflict inside a FCS is said to be free-choice. In particular, T_S is a FCS.

Definition 4 (Transition system). *A transition system is a 4-tuple $A = (S, \Sigma, \rightarrow, s_{in})$, where S is a set of states, Σ is an alphabet of symbols, $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation and s_{in} is the initial state.*

With an abuse of notation, we denote by $s \xrightarrow{e} s', s \rightarrow s', s \rightarrow, \rightarrow s, \ldots$, different facts about the existence of a transition with certain properties.

A path p in a transition system is a sequence of transitions $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \rightarrow \cdots \rightarrow s_n \xrightarrow{e_n} s_{n+1}$, such that the target state of each transition is the source state of the next transition. A path with multiple transitions can also be denoted by $s \xrightarrow{\sigma} s'$, where σ is the sequence of symbols in the path.

3 Schedules

Scheduling of a PN imposes the existence of an additional control mechanism for the firing of enabled transitions. For every marking, a scheduler defines the set of *fireable* transitions as a subset of the enabled transitions. The composite system (PN+scheduler) proceeds from state to state by firing fireable transitions.

Definition 5 (Sequential schedule). *Given a Petri net $N = (P, T, F, M_0)$, a* sequential schedule *of N is a transition system $Sch = (S, T, \rightarrow, s_0)$ with the following properties:*

1. *S is finite and there is a mapping $\mu : S \rightarrow [M_0\rangle$, with $\mu(s_0) = M_0$.*
2. *If transition t is fireable in state s, with $s \xrightarrow{t} s'$, then $\mu(s)[t\rangle\mu(s')$ in N.*
3. *If t_1 is fireable in s, then t_2 is fireable in s if and only if $t_2 \in \mathsf{FCS}(t_1)$.*
4. *t is fireable in s_0 if and only if $t \in T_S$.*
5. *For each state $s \in S$, there is a path $s \xrightarrow{\sigma} s' \xrightarrow{t}$ for each $t \in T_S$.*

Property 2 implies a trace containment for Sch and N (any feasible trace in the schedule is feasible in the original PN). Property 3 indicates that one FCS is scheduled at each state. The FCS scheduled at s_0 is the set of source transitions (property 4). Finally, property 5 denotes the fact that any input event from the environment will be eventually served.

Given a sequential schedule, a state s is said to be an *await state* if only source transitions are fireable in s. An await state models a situation in which the system is "sleeping" and waiting for the environment to produce an event.

Intuitively, scheduling can be deemed as a game between the scheduler and the environment. The rules of the game are the following:

- The environment makes a first move by firing any of the source transitions (property 4 of definition 5).
- The scheduler might pick up any of the enabled transitions to fire (property 3) with two exceptions:
  (a) it has no control over choosing which of the source transitions to fire and
  (b) it cannot resolve choice for data-dependent constructs (which are described by free-choice places).

  In cases (a) and (b) the scheduler must explore all possible branches during the traversal of the reachability space, i.e. fire all the transitions from the same FCS. However it can decide the moment for serving the source transitions or for resolving a free-choice, because it can *finitely* postpone these by choosing some other enabled transitions to fire.

The goal of the game is to process any input from the environment (property 5) while keeping the traversed space finite (property 1). In case of success the result is to both classify the original PN as schedulable and derive the set of states (schedule) that the scheduler can visit while serving an arbitrary mix of source transitions. Under the assumption that the environment is sufficiently slow, the schedule is an upper approximation of the set of states visited during real-time operation.

The notion of sequential schedule is illustrated in Figures 3 and 4. Figure 3 shows two non-schedulable specifications and parts of their reachability spaces. The impossibility to find a schedule for the PN in Fig. 3(a) stems from the inability of a scheduler to control the firing of source transitions. A cyclic behavior in this PN is possible only with correlated input rates of transitions a and b. On the other hand, the PN in Fig. 3(b) is non-schedulable because of the lack of control on the outcome of free-choice resolution for the place $p1$.

Figure 4(a) presents an example of arbitration with two processes competing for the same resource (place p_0). The schedule for this specification is given in Fig. 4(b), where await states are shown by shadowed rectangles. Note that the scheduler makes a smart choice on which one among the concurrently enabled transitions a, d or f fires in the state $\{p_4, p_5\}$, by first scheduling transition f to release the common resource p_0 as quickly as possible.

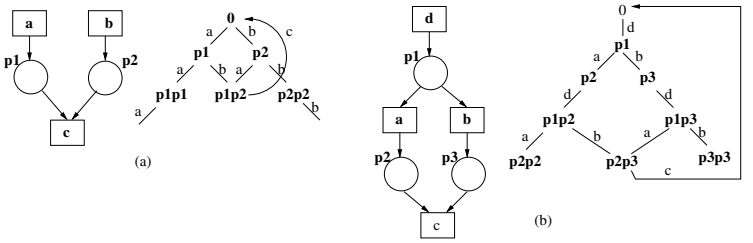


Fig. 3. Non-schedulable PNs

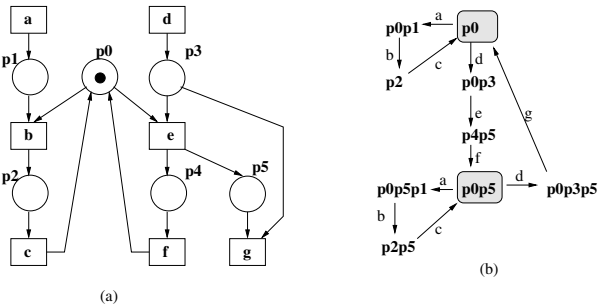


Fig. 4. Processes with arbitration

3.1 Single-Source Schedules: Rationale

As discussed in Section 1, the proposed strategy synthesizes a set of tasks that *serve* the inputs events produced by the environment and may share common data structures in the system. Therefore, their interaction must be consistent, independent of the occurrence order of the external events.

The concept of task in the reactive system corresponds to the concept of single-source schedule (SSS) in our formal model. A SSS is a sequential schedule associated to a single source transition. Each SSS serves only one input channel as if other source transitions were never produced by the environment. In that way a SSS gives a projection of the scheduler activity in which only one source transitions is fireable.

Given a set of SSSs, we want to check whether it can implement the specification of the system. For that, we need to calculate their composition and check that it fulfills the properties of *sequential schedule* (see definition 5).

The rationale behind generation of SSSs in first place rather than constructing a sequential schedule is the following:

- Lower complexity for the generation of SSSs. The size of a sequential schedule can be exponentially larger than the size of the set of SSSs.
- SSSs give a natural decomposition of a sequential schedule which is beneficial for implementation as ISRs on an RTOS.

– A scheduler that behaves according to SSSs has a uniform response for firings of the same source transitions, since each SSS often has just a single await state. This uniformity can be exploited during code generation and provides potentially smaller code size due to the higher probability for sharing pieces of code.

3.2 Single-Source Schedules: Definition and Composition

Definition 6 (Single-source schedule). *Given a Petri net $N = (P, T, F, M_0)$, a* single-source schedule *of N with the transition $a \in T_S$ is a sequential schedule of $N \setminus (T_S \setminus \{a\})$*

Next, sequential composition is defined. The intuitive idea behind this composition is as follows. Each transition system represents a task associated to a source transition. When a task is active, it cannot be preempted, i.e. only events from that task can be issued. A task can only be preempted when it is waiting for an event from the environment (source transition). The composition builds a system that can serve all the events of the environment sequentially.

Definition 7 (Sequential composition). *Let $N = (P, T, F, M_0)$ be a Petri net and $\mathcal{X} = \{SSS(t_i) = (S_i, T_{t_i}, \rightarrow_i, s_{0_i}) \mid t_i \in T_S\}$ be a set of SSSs of N. The sequential composition of $\mathcal{X}$ is a transition system $A = (S, T, \rightarrow, s_0)$ defined as follows:*

– $s_0 = (s_{0_1}, \ldots, s_{0_k})$
– *$S \subseteq S_1 \times \cdots \times S_k$ is the set of states reachable from s_0 according to $\rightarrow$. A state is called an* await *state if all its components are await states in their corresponding SSS.*
– *For every state $s = (s_1, \ldots, s_k)$,*
  - *if s is an await state, then the set of fireable transitions from s is the set of source transitions, i.e. $(s_1, \ldots, s_i, \ldots, s_k) \xrightarrow{t_i} (s_1, \ldots, s'_i, \ldots, s_k)$ in A if and only if $s_i \xrightarrow{t_i} s'_i$ in $SSS(t_i)$.*
  - *if s is not an await state, there is one and only one[1] state component s_i of s such that s_i is not an await state in $SSS(t_i)$. Then the set of fireable transitions from s is the set of fireable transitions from s_i in $SSS(t_i)$, i.e. $(s_1, \ldots, s_i, \ldots, s_k) \xrightarrow{t} (s_1, \ldots, s'_i, \ldots, s_k)$ in A if and only if $s_i \xrightarrow{t} s'_i$ in $SSS(t_i)$.*

Figure 5 depicts the sequential composition of two SSSs obtained from the PN in Fig. 4. The shadowed circles correspond to await states. Initially both SSSs are in await states. Thus, only source transitions a and d are fireable in state 00 of the composition. The firing of any of them, e.g. d, moves the corresponding SSS from the await state and forces the composition to proceed according to the chosen $SSS(d)$ until a new await state is reached (state 3 of $SSS(d)$). In the corresponding state of the composition (state 03) both state components are await states and, therefore, both source transitions a and d are fireable again.

[1] This claim can be easily proved by induction from the definition of $\rightarrow$ and from the fact that s_0 is an await state.

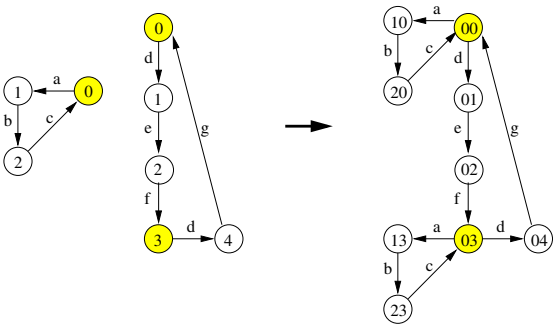


Fig. 5. Two single-source schedules and their sequential composition

Definition 8 (Sequential independence). *Given a Petri net* $N = (P, T, F, M_0)$*, a set of single-source schedules* $\mathcal{X}$ *is* sequentially independent *if its sequential composition is isomorphic to a sequential schedule of* N*.*

One can easily check that the sequential composition in Fig. 5 is isomorphic to the sequential schedule in Fig. 4(b) and, therefore, the set $\{\mathsf{SSS}(a), \mathsf{SSS}(b)\}$, is sequentially independent.

From the definition of SSS, it follows that the existence of a sequential schedule implies the existence of SSSs (once a sequential schedule has been obtained all SSSs can be immediately derived by using the sub-graphs in which only one source transition fires). Moreover, Definition 8 indicates that sequential independence for a set of SSSs is a sufficient condition for the existence of a sequential schedule. In fact, it even gives a constructive way for deriving such a schedule by using the sequential composition of SSSs. For this reason, checking the independence of a set of SSSs is a key issue in the suggested approach.

3.3 Checking Sequential Independence

Given a Petri net N and a set $\mathcal{X}$ of single-source schedules of N, checking their independence can be done as follows:

1. Build the sequential composition A of $\mathcal{X}$.
2. Check that A is a sequential schedule of N, according to Definition 5.

This approach is computationally expensive because it requires to derive explicitly the composition of SSSs.

We next propose an alternative way for checking independence of SSSs that does not require to calculate their composition. Let us consider the case in which the SSSs are not independent, resulting in a failure to find an isomorphic sequential schedule Sch for A. Let us consider paths from the initial states of A and Sch, where Sch mimics A and keeps track of the reachable markings in the Petri net. If there is no independence, there will be two paths that lead to states s and s' in A and Sch, respectively, in which some transition t is enabled in s but

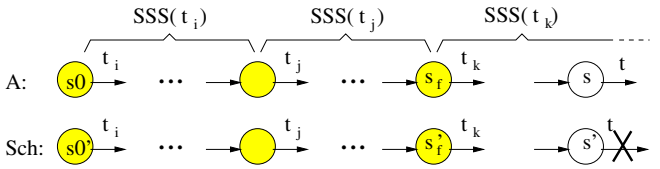


Fig. 6. Matching SSS composition with a sequential schedule

not enabled in s', i.e. the Petri net cannot simulate the sequential composition of SSSs. Figure 6 shows the structure of the paths, where shadowed circles denote await states.

In the last await state s_f before s, $\mathsf{SSS}(t_k)$ is chosen to proceed in the composition by firing transition t_k. The only reason for t not being enabled in state $s' \in Sch$ might come from the "interference" of the execution of the schedules $\mathsf{SSS}(t_i)$ and $\mathsf{SSS}(t_j)$ preceding s_f with $\mathsf{SSS}(t_k)$. Simply speaking, $\mathsf{SSS}(t_i)$ and $\mathsf{SSS}(t_j)$ must consume tokens from some place p in the pre-set of t. This leads to the idea of applying marking equations for the check of SSS independence. It is known that self-loops introduce inaccuracy in calculating the fireable transitions by using the marking equations. For the rest of this section we will assume that a specification is provided as a PN without self-loops [2]. The following hierarchy of notions is used for the formulation of independence via marking properties:

- For $\mathcal{X} = \{\mathsf{SSS}(t_i) \mid t_i \in T_S\}$ and given place p
  - For $\mathsf{SSS}(t_i)$ with set of states S_{t_i} and set of await states $S^a_{t_i}$
    - For state $s \in S_{t_i}$ let $change(p, s) = \mu(s_0)(p) - \mu(s)(p)$, i.e. the difference in token counts for place p between markings corresponding to initial state of $\mathsf{SSS}(t_i)$ and state s.
    - let $\mathsf{SSS}_change(p, t_i) = \max_{s \in S_{t_i}} change(p, s)$, i.e. the maximal change in token count for place p in markings corresponding to states of $\mathsf{SSS}(t_i)$ with respect to the initial marking.
    - let $await_change(p, t_i) = \max_{s \in S^a_{t_i}} change(p, s)$, i.e. the maximal change in token count for place p in markings corresponding to await states of $\mathsf{SSS}(t_i)$ with respect to the initial marking.
  - let $worst_change(p, t_i) = \sum_{t_j \neq t_i, t_j \in T_S} await_change(p, t_j)$, i.e. the sum of $await_change$ for all SSS except for $\mathsf{SSS}(t_i)$

Here is the semantics of the introduced notions:

- $\mathsf{SSS}_change(p, t_i)$ shows how much the original token count for place p deviates while executing the single source schedule $\mathsf{SSS}(t_i)$. If $\mathsf{SSS}(t_i)$ started from the initial marking with a number of tokens in p less than $\mathsf{SSS}_change(p, t_i)$ then $\mathsf{SSS}(t_i)$ would deadlock due to a lack of tokens to fire some transition in the post-set of p.

[2] This requirement does not impose restrictions because any PN with self-loops can be transformed into a self-loop-free PN by inserting dummy transitions.

- $await_change(p, t_i)$ gives a quantitative measure of the influence of $\mathsf{SSS}(t_i)$ on the other schedules. Indeed, as await states are the only points where a scheduler switches among interrupt service routines ($\mathsf{SSS}s$), the change in PN markings due to the execution of $\mathsf{SSS}(t_i)$ is fully captured by the markings of await states, where $await_change(p, t_i)$ gives the worst possible scenario.
- $worst_change(p, t_i)$ generalizes the notion of $await_change(p, t_i)$ to the set of all SSSs except for the chosen $\mathsf{SSS}(t_i)$. The execution of other SSSs has a cumulative influence on $\mathsf{SSS}(t_i)$ expressed by $worst_change(p, t_i)$.

The following theorem establishes a bridge between the sequential independence of SSS and the firing rules in Petri nets when the schedules are executed.

Theorem 1. *A set of single source schedules* $\mathcal{X} = \{\mathsf{SSS}(t_i) \mid t_i \in T_S\}$ *derived from a self-loop free PN* $N = (P, T, F, M_0)$ *is sequentially independent if and only if* $\forall p \in P$ *and* $\forall \mathsf{SSS}(t_i) \in \mathcal{X}$ *the following inequality is true:*

$$M_0(p) - worst_change(p, t_i) - \mathsf{SSS}_change(p, t_i) \geq 0 \qquad (IE.1)$$

Proof. $\Rightarrow$. Suppose that $\mathcal{X}$ is sequentially independent but there exists a place p for which inequality IE.1 is not satisfied. Sequential independence implies the existence of a sequential schedule isomorphic to the composition of $\mathcal{X}$.

In the set of states of the sequential composition of $\mathcal{X}$ let us choose an await state $s = (s_1, \ldots, s_k)$, such that for any $\mathsf{SSS}(t_j), t_j \neq t_i$ the corresponding await component s_j of s is chosen to maximize the token consumption in place p, while s_i is chosen to be the initial state of $\mathsf{SSS}(t_i)$. From the choice of state s follows that by reaching s in the composition, the corresponding marking for place p equals to $M_0(p) - worst_change(p, t_i)$. Let us execute $\mathsf{SSS}(t_i)$ from s. By the definition of $\mathsf{SSS}_change(p, t_i)$ there is a state $s_i' \in \mathsf{SSS}(t_i)$ such that the token count for place p in the marking corresponding to s_i' reduces by $\mathsf{SSS}_change(p, t_i)$ with respect to the initial marking from which $\mathsf{SSS}(t_i)$ starts. From this follows that if $M_0(p) - worst_change(p, t_i) - \mathsf{SSS}_change(p, t_i) < 0$ then in the sequential schedule isomorphic to the sequential composition of $\mathcal{X}$ it would be impossible to fire some transition t that enters state s', where $s' = (s_1, \ldots, s_i', \ldots, s_k)$. The latter contradicts the isomorphism between the composition and the sequential schedule

$\Leftarrow$. Suppose that inequality IE.1 is satisfied but $\mathcal{X}$ is not sequentially independent. In a set of all sequential schedules let us choose the schedule Sch that is isomorphic to the largest subpart of the sequential composition A, i.e. if a mismatch like in Fig. 6 is found by simulating Sch and A then there does not exist any other sequential schedule with state s'' isomorphic to s and capable of firing transition t. Let us rearrange the sequence in Fig. 6 by first executing the schedules other than $\mathsf{SSS}(t_i)$ and let s_f be the first await node in which $\mathsf{SSS}(t_i)$ is chosen. Then the token count for a place p in the marking corresponding to s_f is larger than $M_0(p) - worst_change(p, t_i)$. By definition the execution of $\mathsf{SSS}(t_i)$ cannot reduce it more than by $\mathsf{SSS}_change(p, t_i)$. Then due to the validity of IE.1 when state s' is reached in $\mathsf{SSS}(t_i)$, transition t cannot lack tokens in p needed for its enabling. The case of Fig. 6 is impossible.

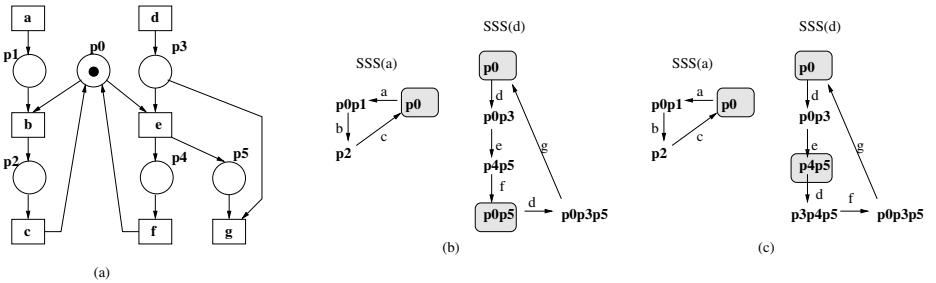


Fig. 7. Process with arbitration and its single source schedules

A sufficient condition for checking the sequential independence can now be derived.

Corollary 1. *A set of single source schedules* $\mathcal{X} = \{SSS(t_i)\}$ *is sequentially independent if for any marking* M *corresponding to an await state* s *of* $SSS(t_i)$ $(M = \mu(s))$ *we have* $\forall p : M(p) \geq M_0(p)$.

Proof. The proof follows from inequality IE.1 by taking into account two observations:

- If for any marking M of the await state s $M(p) \geq M_0(p)$, then $worst_change(p, t_i) \leq 0$.
- The ability of any $\mathsf{SSS}(t_i)$ to be executed from M_0 means that for any place p, $M_0(p) - \mathsf{SSS}_change(p, t_i) \geq 0$. Note that this captures the case of arbitrary PNs (not self-loop free only).

We illustrate the suggested approach with the example in Fig. 7. The two different sets of SSSs are shown in Fig. 7(b,c). The only place shared by both $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ is the place p_0. We can immediately infer the irrelevance of other places with respect to independence violation. Checking the marking count for p_0 in $\mathsf{SSS}(d)$ in Fig. 7(b) gives the following results: $worst_change(p_0, d) = 0$ (p_0 is marked in both await nodes of $\mathsf{SSS}(d)$) and $\mathsf{SSS}_change(p_0, d) = 1$ due to the consumption of p_0 in non-await states of $\mathsf{SSS}(d)$ (see the marking $\{p_4, p_5\}$ e.g.). From similar considerations: $worst_change(p_0, a) = 0$ and $\mathsf{SSS}_change(p_0, a) = 1$. It is easy to see that under the initial marking $M_0(p_0) = 1$ inequality IE.1 is satisfied for both $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$. This is in full correspondence with the conclusion about the sequential independence of $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ that was derived earlier through the explicit construction of their composition (see Fig. 5).

Reversing the order of firing for transitions d and f in $\mathsf{SSS}(d)$ from Fig. 7(c) results in $worst_change(p_0, d)$ increasing to 1 (in the await state $\{p_4, p_5\}$ place p_0 is unmarked). The latter leads to the violation of inequality IE.1 for $\mathsf{SSS}(a)$ and tells about the dependency between $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ from Fig. 7(c). Note that the same result could be immediately concluded by observing await states of $\mathsf{SSS}(d)$ and applying Corollary 1.

From the above example it follows that from the same specification one can obtain independent and dependent sets of SSSs. In case an independent set exists, finding it can be computationally expensive, since an exhaustive exploration of the concurrency in all SSSs may be required. In practice, we suggest to use a "try and check" approach in which a set of SSSs is derived and, if not independent, a sequential schedule is immediately constructed (if possible). This design flow for scheduling is illustrated by Fig. 8.

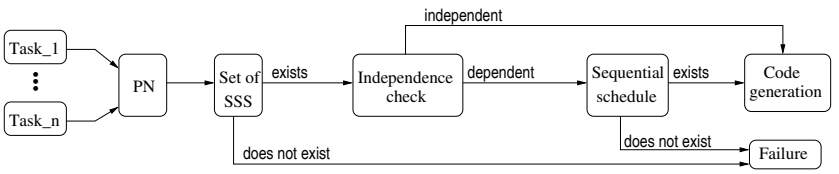


Fig. 8. Design flow for quasi-static scheduling

3.4 Termination Criteria

Single-source schedules are derived by exploring the reachability graph of a Petri net with source transitions. Unfortunately, this graph is infinite.

Next, we discuss conservative heuristic approaches to prune the exploration of the reachability space while constructing a schedule. Conservatism refers to the fact that schedules may not be found in cases in which they exist. Our approach attempts to prune the state space when the search is done towards directions that are qualified as *non-promising*, i.e. the chances to find a valid schedule are remote. The approach is based on defining the notion of *irrelevant marking*. This definition is done in two steps: 1) bounds on places are calculated from the structure of the Petri net and 2) markings are qualified as irrelevant during the exploration of the state space if they cover some preceding marking and exceed the calculated bounds. Note that the property of irrelevance is not local and depends on the pre-history of the marking.

Definition 9 (Place degree). *The degree of a place p is defined as:*

$$degree(p) = \max(M_0(p), \max_{t \in {}^\bullet p} F(t,p) + \max_{t \in p^\bullet} F(p,t) - 1)$$

Place degree intuitively models the "saturation" of p. If the token count of p is $\max F(p,t)$ or more, then adding tokens to p cannot help in enabling output transitions of p. By the firing of a single input transition of p at most $\max F(t,p)$ tokens can arrive, which gives the expression for place degree shown in Definition 9.

Definition 10 (Irrelevant marking). *A marking M is called* irrelevant *with respect to a reachability tree rooted in initial marking M_0, if the tree contains marking M_1 such that:*

- *M is reachable from M_1,*
- *no place has more tokens in M_1 than in M, and*
- *for every place p at which M has more tokens than M_1, the number of tokens in M_1 is equal to or greater than degree(p).*

The example in Fig. 9 illustrates the crucial difference between the approaches targeted to pre-defined place bounds and irrelevant markings.

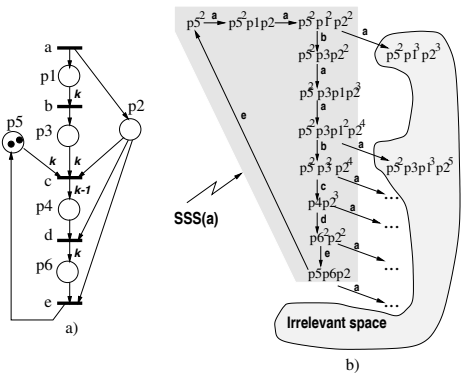


Fig. 9. Constraining the search space by irrelevance criterion

The maximal place degree in PN from Fig. 9(a) is k. This information is the best (as far as we know) one can extract from the PN structure about place bounds. The predefined upper bounds for places should be chosen to exceed place degrees. In fact, the higher place degrees are, the higher upper bounds are expected. Suppose that based on this rationale the upper bounds are chosen as maximal place degree multiplied by some constant margin.

Let us assume for our example that place bounds are assigned to be $2k-1$ and consider the PN reachability space when $k = 2$. When the schedule is checked with the pruning based on pre-defined place bounds, any marking that has more than 3 tokens in a place should be discarded. Clearly no schedule could be found in that reachability space because after a, a, b, a occurs, the only enabled transition is a, but its firing produces 4 tokens in place $p2$ (see the part of reachability graph shown in Fig. 9(b), where superscripts near places show the number of tokens the place has under the current marking). The search fails.

The irrelevance criterion handles this problem more graciously. It guides the search for the "proper" direction in the reachability space by avoiding the irrelevant markings. The first guidance is given when marking $\{p_5^2, p_1^2, p_2^2\}$ is reached. In that marking one need to choose which transitions a or b to fire from the enabled set. The firing of a however produces the marking $\{p_5^2, p_1^3, p_2^3\}$ which is irrelevant because it covers $\{p_5^2, p_1^2, p_2^2\}$, where places p_1 and p_2 are already saturated. Therefore transition b should be chosen to fire. After this, a fires two times, resulting in the marking $\{p_5^2, p_3, p_1^2, p_2^4\}$. Note that even though the

place degree for p_2 is exceeded in this marking, the marking is not irrelevant because in all the preceding markings containing p_3, p_1 is not saturated. From this marking the system is guided to fire b because the firing of a again would enter the irrelevant space (see Fig. 9(b)). Finally this procedure succeeds and finds a valid SS schedule.

Though pruning the search by using irrelevance seems a more justified criterion than by using place bounds, it is not exact for general PNs. There exist PNs for which any possible schedule enters the irrelevant space. This is due to the fact that for general PNs accumulating tokens in choice places after their saturation could influence the resolution of choice (e.g., by splitting token flows in two choice branches simultaneously). If for any choice place p in PN either at most one of transitions in $p^\bullet$ is enabled (unique choice) or every transition in $p^\bullet$ is enabled (free-choice) then adding tokens to p does not change the choice behavior of a PN. This gives the rationale behind the conjecture that *the irrelevant criterion is exact for PNs with choice places that are either unique or free-choice*. However we are unable either to prove the exactness of this criterion or to find a counterexample for that. This issue is open for the moment.

4 Algorithm for Schedule Generation

This section presents an algorithm for computing a sequential schedule. It can also be used to compute a single-source schedule for a source transition t_i, if it takes as input the net in which all the source transitions except t_i are deleted (see Definition 6). Finally, a sequential program is generated from the resulting schedule by the procedure described in Sect. 4.2.

4.1 Synthesis of Sequential Schedules

Given a PN N, the scheduling algorithm creates a directed tree, where nodes and edges are associated with markings and transitions of N respectively. In the sequel, $\mu(v)$ denotes the marking associated with a node v of the tree, while $T([v,w])$ denotes the transition for an edge $[v,w]$. Initially, the root r is created and $\mu(r)$ is set to the initial marking of N. We then call a function $\texttt{EP}(r,\ r)$, shown in Figure 10(a). If this function returns successfully, the post-processing is invoked to create a cycle for each leaf. The resulting graph represents a sequential schedule $(S,T,\rightarrow,r)$, where S is the set of nodes of the graph, T is the set of transitions of N, and $\rightarrow$ is given by $v \stackrel{T([v,w])}{\rightarrow} w$ for each edge $[v,w]$.

EP takes as input a leaf v of the current tree and its ancestor *target*. We say that a node u is an *ancestor* of v, denoted by $u \leq v$, if u is on the path from the root to v. If in addition $u \neq v$, u is a *proper ancestor* of v, denoted by $u < v$. EP creates a tree rooted at v, where each node x is associated with at most one FCS enabled at $\mu(x)$. The goal is to find a tree at the root v with two properties. First, each leaf has a proper ancestor with the same marking. Second, each non-leaf x is associated with an FCS so that for each transition t of the FCS, x has a child y with $T([x,y]) = t$. If such a tree is contained in the

```
function EP(v, target)        // returns (status, ap, ep)
    ap ← 0, ep ← UNDEF, FCS(v) ← φ;
    if(termination conditions hold) return (0, 0,UNDEF);
    if(∃u : u < v and μ(u) = μ(v)) return (1, 0, u);
    for(each FCS F enabled at μ(v))
        if(v = r and F ≠ T_S) continue;     // r is the root.
        if(F = T_S) current_target ← v;
        else current_target ← target;
        (status, ap_F, ep_F) ← EP_FCS(F, v, current_target);
        if(status = 0) continue;
        if(ap_F = 1)
            FCS(v) ← F, return (1, 1, ep_F);
        if(ep_F ≤ current_target)
            FCS(v) ← F, return (1, 0, ep_F);
        if(ep_F < ep)
            FCS(v) ← F, ap ← ap_F, ep ← ep_F;
    if(FCS(v) = φ) return (0, ap, ep);
    else return (1, ap, ep);
```

(a)

```
function EP_FCS(F, v, target)
        // returns (status, ap_F, ep_F)
    ap_F ← 0, ep_F ← UNDEF,
    current_target ← target;
    for(each transition t of F)
        create a node w and an edge [v, w];
        T([v, w]) ← t;
        μ(w) ← the marking obtained by
                firing t at μ(v);
        (status, ap, ep) ← EP(w, current_target);
        if(status=0) return (0, ap_F, UNDEF);
        if(ap = 1 or FCS(w) = T_S)
            ap_F ← 1, current_target ← v;
        if(ap_F = 0 and v < ep)
            return (0, 0, UNDEF);
        if(ep ≤ v) ep_F ← min(ep_F, ep);
        if(ep_F ≤ target) current_target ← v;
    return (1, ap_F, ep_F);
```

(b)

Fig. 10. The two main functions called in computing a sequential schedule

one created by EP, we say that EP succeeds at v. FCS's are associated so that the conditions given in the definition of sequential schedules (Definition 5) are satisfied, which will be elaborated next.

EP returns three values, denoted by *status(v)*, $ap(v)$, and $ep(v)$. There are two terminal cases, given in the third and fourth lines of the code in Fig. 10(a), for which the returned values are presented respectively. Suppose that v does not fall into the terminal cases. *status(v)* is a boolean variable, which is 1 if and only if EP succeeds at v. The other two values are meaningful only if *status(v)* is 1. $ap(v)$ is a boolean variable, which is 1 if and only if v has a path to an await node in the created tree such that for each edge on the path, say $[x, y]$, an FCS is associated with x and $T([x, y])$ is in the FCS. A node is said to be *await* if it is associated with an FCS and this is the set of source transitions T_S.

$ep(v)$ is an ancestor u of v for which there exists an FCS enabled at $\mu(v)$ that satisfies the following three conditions. First, for each transition t of the FCS, a child w has been created with $T([v, w]) = t$ and $\mu(v)[t\rangle\mu(w)$. Second, for each such w, *status(w)*$= 1$ and either $ap(w) = 1$ or $ep(w) \leq v$. Third, u is the minimum among $ep(w)$ for all w such that $ep(w) \leq v$, i.e. the one closest to the root r. If no $ep(w)$ is an ancestor of v, or if there is no FCS that satisfies these conditions, $ep(v)$ is set to UNDEF. Intuitively, if *ep(v)* is not UNDEF, it means that there exists an FCS enabled at $\mu(v)$ with the property that for each transition t of the FCS, if $ep(w) \leq v$ holds for the corresponding child w, there is a sequence of transitions starting from t that can be fired from $\mu(v)$ and the marking obtained after the firing is $\mu(u)$. Further, at each marking obtained during the firing of the sequence, there is an FCS enabled at the marking that satisfies this property. If there exists such an FCS at v, EP further checks if there is one that also satisfies $ep(v) \leq target$. If this is the case, EP associates one of them with v, which is denoted by $FCS(v)$ in the algorithm. Otherwise, EP associates any FCS with the conditions above. If no such FCS exists, no FCS is associated and $FCS(v)$ is set to empty.

To find such an FCS, EP calls a function EP_FCS for each FCS enabled at $\mu(v)$. The enabled FCS's are sorted in EP before calling EP_FCS, so that T_S is positioned at the end of the order. This heuristic tries to minimize the number of await nodes introduced in a schedule. The exception of this rule is at the root r, in which EP_FCS is called only for T_S. This ensures the property 4 given in the definition of sequential schedules. If EP succeeds at the root r, we call the post-processing to create a schedule and terminate. Otherwise, we report no schedule and terminate.

The post-processing consists of two parts. First, we retain only a part of the created tree that are used in the resulting schedule, and delete the rest. The root is retained, and a node w is retained if its parent v is retained and the transition $T([v, w])$ is in $FCS(v)$. Second, a cycle is created for each leaf w of the retained portion of the tree, by merging w with its proper ancestor u such that $\mu(u) = \mu(w)$. By construction, such a u uniquely exists for w. The graph obtained at the end is returned. It is shown that this algorithm always finds a schedule, if there exists one in the space defined by the terminate conditions employed in EP.

The resulting graph may have more than one node with the same marking and FCSs associated with these nodes may not be the same. The freedom of associating different FCSs at nodes with the same marking allows the scheduler to explore the larger solution space, and thus the algorithm does not commit to the same FCS for a given marking.

4.2 Code Generation

The code generation algorithm takes a graph of a sequential schedule and synthesizes code [11]. We briefly illustrate it in this section.

The algorithm proceeds in three steps. First, we traverse the schedule Sch to identify a set of sub-trees that *covers* Sch, i.e. for each node v of Sch, there exists a tree that contains a node v' with $FCS(v) = FCS(v')$. We say that v *corresponds* to v'. Our procedure finds a minimal set with an additional property that for each v, there is exactly one v' that v corresponds to.

The second step generates code for each tree. This code is made of two types. One is the code for realizing the control flow statements. For example, `if-then-else` is introduced at a node with multiple child nodes. Also, `switch` and `goto` are used to jump from each leaf of the tree to the root of another tree. For this purpose, variables are introduced for places and the markings in the tree are represented with them. This and the correspondence information obtained in the first step are used to implement the jump correctly. The other type of code is operations executed at each transition from one node to another. In our applications, the Petri net is generated so that each transition is annotated with a sequential program. This program is copied in the generated code.

The third step is concerned with channels between processes that have been merged into a single schedule. For each such channel, we define a circular buffer and replace write and read operations for the channel that appear in the generated code with operations on the buffer. The size of the buffer can be statically

identified as the upper bound found in the schedule. If the buffer has size 1, it is substituted by a single variable.

5 Experimental Results

We used as our test system an MPEG-2 video decoder developed by Philips (see [14]) and shown in Fig. 11. Processes Thdr and Tvld parse the input video stream; Tisiq and Tidct implement spatial compression decoding; TdecMV, Tpredict and Tadd are responsible for decoding temporal compression and generating the image; Tmemory, TwriteMB, TmemMan and Toutput manage the frame store and produce the output to be sent to a visualization device. Communication is by means of channels that can handle arbitrary data types. Philips used approximately 7700 lines of code to describe the 11 processes, and 51 channels. An average of 16 communication primitives per process are used to transfer data through those channels.

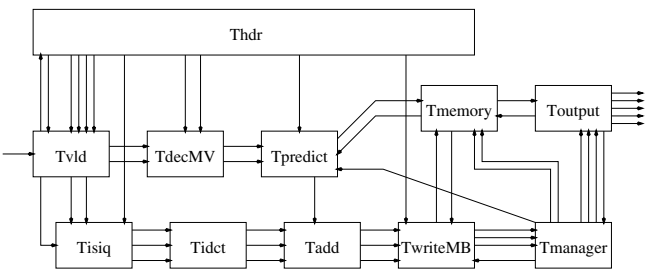


Fig. 11. MPEG-2 video decoder block diagram

In the original implementation, all processes were scheduled at run time. Our objective was to reduce scheduling overhead by merging processes into quasi-statically scheduled ones. We focused our attention on five processes: Tisiq, Tidct, TdecMV, Tpredict and Tadd. They consist of about 3000 lines of code and account for more than half of all communications occurring in the system. The inputs to these five processes from the rest of the system are correlated, and thus they cannot be treated as independent inputs. Instead of modeling the correlation explicitly using additional processes, we introduced a single input that triggers the five processes to react to this input. As a result, our procedure generated a single source schedule for this trigger input.

The Petri net generated from the FlowC specification has 115 places, 106 transitions and 309 arcs. Our algorithm generated a single process with the same interface as the original ones, that could be plugged into the MPEG-2 netlist, replacing the original five processes.

An example of how the code is transformed after scheduling is shown in Fig. 12. Figure 12(a) shows a small fragment of code taken from the processes

`Tpredict` and `Tadd`. They both implement a `while` loop during which they exchange some data (a macro-block is written from `Tpredict` to `Tadd`).

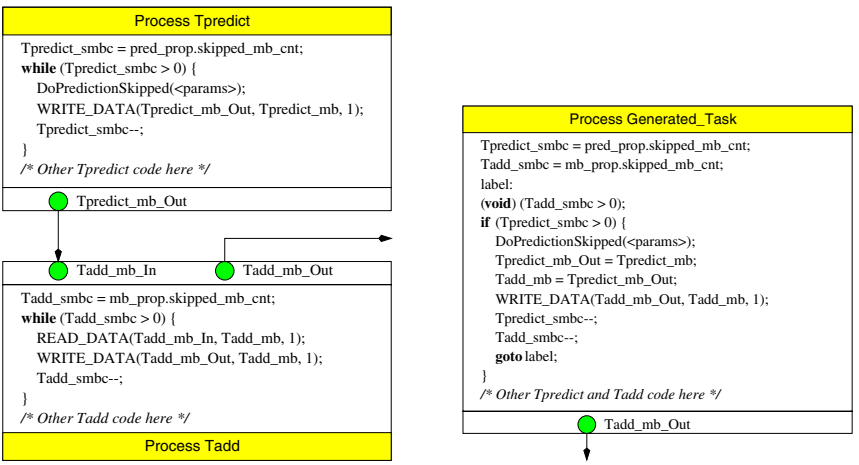


Fig. 12. (a) Example of FlowC specification, (b) Portion of the generated code for the MPEG-2 decoder

On the other hand, Fig. 12(b) shows the same fragment in the generated process, where the two were merged into a single entity. What is generated is a single loop which contains statements from the two original processes. Note that the `WRITE_DATA` and `READ_DATA` statements in processes `Tpredict` and `Tadd` occurring on the channel connecting them have been transformed into assignments to and from a temporary variable (which can be easily eliminated by an optimizing compiler). The `WRITE_DATA` statement in `Tadd` on the output channel is instead preserved as is, and needs to be expanded to match the communication protocol used in the rest of the system (in our case, it is a FIFO).

We compared the performance of the original specification with that of the same system where a single statically scheduled process. In both cases, we removed the processes that manage and implement the memory, but we kept those that parse the input MPEG stream. Both systems received as input a video stream composed of 4 images (1 intra, 1 predicted, 2 bidirectional predicted).

Table 5 summarizes the total execution time on a Sun Ultra Enterprise 450. It also shows the individual contributions of the processes (split among the parser and the five processes that we scheduled together), the test-bench and the operating system. The increase in performance is around 45%. The gain is concentrated in the statically scheduled processes, due to the reduction in the number of FIFO-based communications, and in the operating system due to the reduction in the number of context switches.

Table 1. CPU time, in seconds, of the MPEG-2 example

	Total	**MPEG2**			**Test**	**OS**
		Total	Parser	5Procs	**bench**	
Orig.	7.5	4.66	0.94	3.72	0.27	2.58
QSS	4.1	2.51	0.94	1.57	0.28	1.31

Table 2. CPU time, in seconds, and code size of the five selected processes

	5 Processes				
	Total	**Comp.**	**Int. Comm.**	**Ext. Comm.**	**Code size**
Orig.	3.72	1.01	2.23	0.48	18K
QSS	1.57	0.96	0.13	0.48	24K

Table 5 compares the execution times due to computation and communication of the five processes, both in the original system and in the quasi-statically scheduled one. As expected, computation and external communication are not significantly affected by our procedure. However, internal communication is largely improved: this is because after scheduling we could statically determine that all channels connecting the five considered processes never have more than one element or structure at a time. Therefore, communication is performed by assignment, rather than by using a FIFO or a circular buffer. The table also report the object code size, which increases in the generated single task with respect to the 5 separated process: this is due to the presence of control structures representing the static schedule in the synthesized code.

6 Conclusions

This paper proposes a method that bridges the gap between specification and implementation of reactive systems. From a set of communicating processes, and by deriving an intermediate representation based on Petri nets, a set of concurrent tasks that serve input events with minimum communication effort is obtained. This paper has presented a first effort in automating this bridge. Experiments show promising results and encourage further research in the area.

In the future, we expect a more general definition of the concept of schedule, considering concurrent implementations, and a structural characterization for different classes of Petri nets.

Acknowledgements

This work has been partically funded by a grant from Cadence Design Systems and CICYT TIC 2001-2476.

References

1. J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, U. C. Berkeley, 1993. 81
2. J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software. In *Proceedings of the 37th Design Automation Conference*, June 2000. 81
3. E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application Modeling for Singal Processing Systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000. 81
4. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993. 80
5. D. Har'el, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990. 80
6. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 66 Wood Lane End, Hemel Hempstead, Hertfordshire, HP2 4RG, UK, 1985. 80
7. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, August 1974. 80
8. H. Kopetz and G. Grunsteidl. TTP – A protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1), January 1994. 81
9. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, January 1987. 80, 81
10. B. Lin. Software synthesis of process-based concurrent programs. In *35th ACM/IEEE Design Automation Conference*, June 1998. 81
11. C. Passerone, Y. Watanabe, and L. Lavagno. Generation of minimal size code for schedule graphs. In *Proceedings of the Design Automation and Test in Europe Conference*, March 2001. 96
12. M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *36th ACM/IEEE Design Automation Conference*, June 1999. 81
13. K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and et al. Scheduling hardware/software systems using symbolic techniques. In *International Workshop on Hardware/Software Codesign*, 1999. 81
14. P. van der Wolf, P. Lieverse, M. Goel, D. L. Hei, and K. Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proceedings of the* 7^{th} *International Workshop on Hardware/Software Codesign*, May 1999. 97

Data Decision Diagrams for Petri Net Analysis

Jean-Michel Couvreur[1], Emmanuelle Encrenaz[2], Emmanuel Paviot-Adet[2], Denis Poitrenaud[2], and Pierre-André Wacrenier[1]

[1] LaBRI, Université Bordeaux 1
Talence, France
<name>@labri.fr
[2] LIP6, Université Pierre et Marie Curie
Paris, France
<name>@lip6.fr

Abstract. This paper presents a new data structure, the Data Decision Diagrams, equipped with a mechanism allowing the definition of application-specific operators. This mechanism is based on combination of inductive linear functions offering a large expressiveness while alleviating for the user the burden of hard coding traversals in a shared data structure. We demonstrate the pertinence of our system through the implementation of a verification tool for various classes of Petri nets including self modifying and queuing nets.

Topics. Petri Nets, Decision Diagram, System verification.

The design and verification of distributed systems present a scientific and technical challenge that must be met by a combination of techniques that scale up to the complexity of systems produced in industrial applications. Simulation is already a recognized tool within industries dealing with complex systems such as telecommunications or aeronautics. Even if an acceptable degree of confidence can be reached via this technique, exhaustiveness cannot generally be reached due to the number of possible states of these systems.

In the 90's, the electronic industries, in search of tools to increase their confidence level in their final product, adopted the Binary Decision Diagrams (BDDs) [1] as a way to deal with the high complexity of their components. BDD can be viewed as a tree structure: binary variables involved in states are ordered, a set of nodes is associated to each variable and the variable valuations are associated to the arcs between nodes. When implemented, the uniqueness of BDDs, combined with the tree structure, ensures an efficient technique to deal with numerous states [6,4]. Then exhaustiveness could be handled, even with billions and billions of states to verify.

The BDDs expressive power is large enough to deal with a large class of finite state system. Even dynamic systems can be verified using such a technique [5]. Therefore, other domains, like parallel system verification, tried to use BDDs to verify complex systems [8]. Since the number of variables of the studied system is a critical parameter, numerous BDD-like structures were created in order to adapt the tree structure to particular needs [7,2].

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 101–120, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Like BDDs, the shared-tree structures are usually stuck to a precise interpretation. Therefore, dealing with states of a new kind of model usually leads to the design of new kinds of structures or to new kinds of operators on existing structures.

In this paper, a new tree structure, the Data Decisions Diagrams (DDDs), is introduced. Our goal is to provide a flexible tool that can be easily adapted to any kind of model and that offers the same storage capabilities as the BDD-like structures. Unlike previous works on the subject, operators on the structure are not hard-coded, but a class of operators, called homomorphisms, are introduced to allow transition rules coding. A special kind of homomorphisms, called inductive homomorphisms, allows to define "local" homomorphisms, i.e. homomorphisms that only use information local to a node in its definition. Together with composition, concatenation, union... operations, general homomorphisms are defined.

In our model, a node is associated to a variable and valuations are associated to arcs. The variables domain is integer, but we do not have to know *a priori* bounds.

Another nice feature of the DDDs is that no variable ordering is presupposed in definition. Moreover, variables are not assumed to be part of all paths. They also can be seen many times along the same path. Therefore, the maximal length of a path in the tree is not fixed. This feature is very useful when dealing with dynamic models like queuing Petri nets, but also when a temporary variable is needed (cf the mechanism we used to code self modifying nets firing rules). Even if a global variable ordering is useful to obtain an efficient storage, the fact that this ordering is not part of the definition introduces a great flexibility when one needs to encode a state. Since we use techniques that have been shown efficient to store set of states, since state coding is very flexible and since operator definition is based on a well-founded theoretical foundation, DDD is a tree structure that can be easily adapted to any kind of computational model.

This paper is structured the following way: Section 2 describes DDDs, the homomorphisms, and gives some hints on the implementation we made in our prototype. Section 3 describes a possible use of DDDs for ordinary Petri nets and some of the most popular extensions (inhibitors arcs, capacity places, reset arcs, self modifying nets, queuing nets). Section 4 gives conclusions with perspectives.

1 Data Decision Diagrams

1.1 Definitions of DDDs

Data Decision Diagrams (DDD) are data structures for representing sets of sequences of assignments of the form $e_1 = x_1; e_2 = x_2; \cdots e_n = x_n$ where e_i are variables and x_i are values. When an ordering on the variables is fixed and the variables are boolean, DDD coincides with the well-know Binary Decision Diagram. If an ordering on the variables is the only assumption, DDDs are the specialized version of the Multi-valued Decision Diagrams representing characteristic function of sets [7,2]. For Data Decision Diagram, we assume no variable

ordering and, even more, the same variable may occur many times in an assignment sequence, allowing the representation of dynamic structures.

Traditionally, decision diagrams are often encoded as decision trees. Figure 1, left-hand side, shows the decision tree for the set of sequences of assignments $A = \{(a = 1; a = 1), (a = 1; a = 2; b = 0), (a = 2; b = 3)\}$. As usual, node 1 stands for accepting terminator and node 0 for non-accepting terminator. Since, there is no assumption on the cardinality of the variable domains, we consider node 0 as the default value. Therefore node 0 is not depicted in the figure.

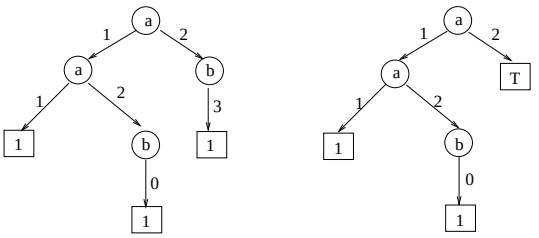


Fig. 1. Two Data Decision Diagrams

Any set of sequences cannot be represented, we thus introduce a new kind of leaf $\top$ for undefined. Figure 1, right-hand side, gives an approximation of the set $A \cup \{(a = 2; a = 3)\}$. Indeed, an ambiguity is introduced since after assigning the first value of variable $a = 2$, one may want to assign both $a = 3$ and $b = 3$. Such assignment sequences cannot be represented in the structure: from a node, only one arc of a given value is allowed.

In the following, E denotes a set of variables, and for any e in E, $\mathrm{Dom}(e)$ represents the domain of e.

Definition 1 (Data Decision Diagram). *The set* $\mathbb{D}$ *of DDDs is defined by* $d \in \mathbb{D}$ *if:*

- $d \in \{0, 1, \top\}$ *or*
- $d = (e, \alpha)$ *with:*
  - $e \in E$
  - $\alpha : \mathrm{Dom}(e) \to \mathbb{D}$*, such that* $\{x \in \mathrm{Dom}(e) \mid \alpha(x) \neq 0\}$ *is finite.*

We denote $e \xrightarrow{a} d$*, the DDD* (e, α) *with* $\alpha(a) = d$ *and for all* $x \neq a$*,* $\alpha(x) = 0$*.*

A straight definition of DDDs equality would lead to a non-unique representation of the empty set (any structure with node 0 as unique terminator). Therefore we introduce the following equivalence relation:

Proposition 1 (Equivalence relation). *Let* $\equiv$ *the relation inductively defined for* $\forall d, d' \in \mathbb{D}$ *as*

- $d = d'$ *or*
- $d = 0$, $d' = (e', \alpha')$ *and* $\forall x \in \mathrm{Dom}(e') : \alpha'(x) \equiv 0$ *or*
- $d = (e, \alpha)$, $d' = 0$ *and* $\forall x \in \mathrm{Dom}(e) : \alpha(x) \equiv 0$ *or*
- $d = (e, \alpha)$, $d' = (e', \alpha')$ *and* $(d \equiv 0) \wedge (d' \equiv 0)$ *or*
- $d = (e, \alpha)$, $d' = (e', \alpha')$ *and* $(e = e') \wedge (\forall x \in \mathrm{Dom}(e) : \alpha(x) \equiv \alpha'(x))$

The relation $\equiv$ *is an equivalence relation*[1].

From now on, a DDD is an equivalence class of the relation $\equiv$. We will use 0 to represent any empty DDD. This induces a canonical representation of DDDs: in a DDD, nodes equivalent to 0 are replaced by the terminator node 0.

An important feature of DDDs is the notion of approximation of assignment sequence sets based on the terminator node $\top$: $\top$ represents any set of assignment sequences. When $\top$ does not appear in a DDD, the DDD represents a unique set of assignment sequences; we say that it is well-defined.

Definition 2 (Well-defined DDD). *A DDD* d *is well-defined if*

- $d = 0$ *or*
- $d = 1$ *or*
- $d = (e, \alpha)$ *where* $\forall x \in \mathrm{Dom}(e) : \alpha(x)$ *is well-defined.*

In other words, $\top$ is the worst approximation of a set of assignment sequences. It induces inductively a partial order which formalizes the notion of approximation: the better-defined relation.

Proposition 2 (Better-defined partial order). *Let the* better-defined *relation, denoted* $\preceq$*, be inductively defined by* $\forall d, d' \in \mathbb{D}$ *as* $d \preceq d'$ *if*

- $d' = \top$ *or*
- $d \equiv d'$ *or*
- $d \equiv 0$, $d' = (e', \alpha')$ *and* $\forall x \in \mathrm{Dom}(e') : 0 \preceq \alpha'(x)$
- $d = (e, \alpha)$, $d' = (e', \alpha')$ *and* $(e = e') \wedge (\forall x \in \mathrm{Dom}(e) : \alpha(x) \preceq \alpha'(x))$

The relation $\preceq$ *is a partial order on equivalent classes of DDDs. Moreover well-defined DDDs are the minimal DDDs.*

In order to generalize operators over assignment sequence sets, we observe that better-defined the operands are, better-defined the result is. This leads to the following definition:

Definition 3 (DDD operator). *Let* f *be a mapping* $\mathbb{D}^n \to \mathbb{D}$*.* f *is an operator on* $\mathbb{D}$ *if* f *is compatible with the partial order* $\preceq$*:*

$$\forall (d_i)_i \in \mathbb{D}^n, \forall (d'_i)_i \in \mathbb{D}^n : (\forall i : d_i \preceq d'_i) \Rightarrow f((d_i)_i) \preceq f((d'_i)_i)$$

[1] Proofs of propositions are omitted since they are simple but tedious. Indeed, for each proposition, one simply has to enumerate each kind of DDD appearing in it and check the properties.

1.2 Operations on DDDs

First, we generalize the usual set-theoretic operations – sum (union), product (intersection) and difference – to sets of assignment sequences expressed in terms of DDDs. The crucial point of this generalization is that all DDDs are not well-defined and moreover that the result of an operation on two well-defined DDDs is not necessarily well-defined. We propose definitions of these set-theoretic operations which produce the best approximation of the result as possible. In particular, when the two operands are well-defined and when the result may be represented by a well-defined DDD, then the following operators produce exactly this one.

Definition 4 (Set operators). *The* sum $+$*, the* product $*$*, the* difference $\setminus$ *of two DDDs are defined inductively as follows:*

$+$	$0 \ \vee (e_2, \alpha_2) \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$0 \ \vee (e_1, \alpha_1) \equiv 0$	0	1	$\top$	(e_2, α_2)
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	(e_1, α_1)	$\top$	$\top$	$(e_1, \alpha_1 + \alpha_2)$ *if* $e_1 = e_2$ $\top$ *if* $e_1 \neq e_2$

$*$	$0 \ \vee (e_2, \alpha_2) \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$0 \ \vee (e_1, \alpha_1) \equiv 0$	0	0	0	0
1	0	1	$\top$	0
$\top$	0	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	0	0	$\top$	$(e_1, \alpha_1 * \alpha_2)$ *if* $e_1 = e_2$ 0 *if* $e_1 \neq e_2$

$\setminus$	$0 \ \vee (e_2, \alpha_2) \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$0 \ \vee (e_1, \alpha_1) \equiv 0$	0	0	0	0
1	1	0	$\top$	1
$\top$	$\top$	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	(e_1, α_1)	(e_1, α_1)	$\top$	$(e_1, \alpha_1 \setminus \alpha_2)$ *if* $e_1 = e_2$ (e_1, α_1) *if* $e_1 \neq e_2$

where, for any $\diamond \in \{+, *, \setminus\}$*,* $\alpha_1 \diamond \alpha_2$ *stands for the mapping in* $\mathrm{Dom}(e_1) \to \mathbb{D}$ *with* $\forall x \in \mathrm{Dom}(e_1) : (\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$*.*

The concatenation operator defined below corresponds to the concatenation of language theory. Nevertheless, the definition takes into account the approximation aspect.

Definition 5 (Concatenation operator). *Let d, d' be two DDDs. The* concatenation $d \cdot d'$ *is inductively defined as follows:*

$$d \cdot d' = \begin{cases} 0 & \text{if } d = 0 \vee d' \equiv 0 \\ d' & \text{if } d = 1 \\ \top & \text{if } d = \top \wedge d' \not\equiv 0 \\ (e, \sum_{x \in Dom(e)} (e \xrightarrow{x} (\alpha(x) \cdot d'))) & \text{if } d = (e, \alpha) \end{cases}$$

The operators respect the definition of operator on DDD (Def. 3) and have the usual properties as commutativity and associativity but $*$ is not associative.

Proposition 3 (Basic operator properties). *The operators $*, +, \setminus, \cdot$ are operators on DDDs. Moreover, $*, +$ are commutative and $+, \cdot$ are associative. Operator $*$ is not associative.*

Thanks to the basic properties of operator $+$, we may denote a DDD (e, α) as $\sum_{x \in Dom(e)} (e \xrightarrow{x} \alpha(x))$. Remark that this sum has a finite number of non null DDDs.

Example 1. Let d_A be the DDD represented in left-hand side of Fig. 1, and d_B the right-hand side one. Notice that any DDD may be defined using the constants 0, 1, $\top$, the elementary concatenation $e \xrightarrow{x} d$ and the operators $+$.

$$d_A = a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} b \xrightarrow{3} 1$$
$$d_B = a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top$$

Let us now detail some computations:

$$\begin{aligned} d_A + a \xrightarrow{2} a \xrightarrow{3} 1 &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \left(b \xrightarrow{3} 1 + a \xrightarrow{3} 1 \right) \\ &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top \\ &= d_B \end{aligned}$$
$$\left(a \xrightarrow{1} 1 * a \xrightarrow{2} 1 \right) * \top = 0 * \top = 0 \neq a \xrightarrow{1} 1 * \left(a \xrightarrow{2} 1 * \top \right) = a \xrightarrow{1} 1 * \top = \top$$
$$d_A \setminus d_B = a \xrightarrow{2} \left(b \xrightarrow{3} 1 \setminus \top \right) = a \xrightarrow{2} \top$$
$$d_B \cdot c \xrightarrow{4} 1 = a \xrightarrow{1} \left(a \xrightarrow{1} c \xrightarrow{4} 1 + a \xrightarrow{2} b \xrightarrow{0} c \xrightarrow{4} 1 \right) + a \xrightarrow{2} \top$$

The next step of our formalization is to introduce homomorphisms over DDDs to allow the definition of complex operations. In the aim of using the previous operators in the homomorphism context, distributivity is a crucial property. However, it must be adapted to take into account the approximation feature.

Proposition 4 (Weak-distributivity). *The product operator $*$ and the concatenation operator $\cdot$ are weakly distributive with the sum operator $+$ and the difference operator $\setminus$ is weakly right distributive with operator $+$:*

$$\forall d_1, d_2, d \in \mathbb{D} : \begin{cases} d * d_1 + d * d_2 \preceq d * (d_1 + d_2) \\ (d_1 \setminus d) + (d_2 \setminus d) \preceq (d_1 + d_2) \setminus d \\ d \cdot d_1 + d \cdot d_2 \preceq d \cdot (d_1 + d_2) \\ (d_1 \cdot d) + (d_2 \cdot d) \preceq (d_1 + d_2) \cdot d \end{cases}$$

The well-foundation of these propositions is that, in the absence of undefined values ($\top$), we regain the usual properties of these binary operators.

1.3 Homomorphisms on DDDs

Our goal is to generalize the notion of homomorphism to DDD taking into account the approximation introduced by $\top$. The classical identity $f(d_1)+f(d_2) = f(d_1+d_2)$ is rewritten using the better-defined relation. One can notice that the weak distributivity that we have just introduced above matches for classical mappings as $d * \mathrm{Id}$, $\mathrm{Id} \setminus d$, $\mathrm{Id} \cdot d$, $d \cdot \mathrm{Id}$ where d is a given DDD and Id is the identity. Another requirement for homomorphisms is to be DDD operators and to map 0 to 0.

Definition 6 (Homomorphism). *A mapping Φ on DDDs is an* homomorphism *if $\Phi(0) = 0$ and*

$$\forall d_1, d_2 \in \mathbb{DD} : \begin{cases} \Phi(d_1) + \Phi(d_2) \preceq \Phi(d_1 + d_2) \\ d_1 \preceq d_2 \Rightarrow \Phi(d_1) \preceq \Phi(d_2) \end{cases}$$

The sum and the composition of two homomorphisms are homomorphisms.

Proposition 5 (Sum and composition). *Let Φ_1, Φ_2 be two homomorphisms. Then $\Phi_1 + \Phi_2$, $\Phi_1 \circ \Phi_2$ are homomorphisms.*

So far, we have at one's disposal the homomorphism $d * \mathrm{Id}$ which allows to select the sequences belonging to the given DDD d; on the other hand we may also remove these given sequences, thanks to the homomorphism $\mathrm{Id} \setminus d$. The two other interesting homomorphisms $\mathrm{Id} \cdot d$ and $d \cdot \mathrm{Id}$ permit to concatenate sequences on the left or on the right side. For instance, a widely used left concatenation consists in adding a variable assignment $e_1 = x_1$ that is denoted $e_1 \xrightarrow{x_1} \mathrm{Id}$. Of course, we may combine these homomorphisms using the sum and the composition.

However, the expressive power of this homomorphism family is limited; for instance we cannot express a mapping which modifies the assignment of a given variable. A first step to allow user-defined homomorphism Φ is to give the value of $\Phi(1)$ and of $\Phi(e \xrightarrow{x} d)$ for any $e \xrightarrow{x} d$. The key idea is to define $\Phi(e, \alpha)$ as $\sum_{x \in \mathrm{Dom}(e)} \Phi(e \xrightarrow{x} \alpha(x))$ and $\Phi(\top) = \top$. A sufficient condition for Φ being an homomorphism is that the mappings $\Phi(e, x)$ defined as $\Phi(e, x)(d) = \Phi(e \xrightarrow{x} d)$ are themselves homomorphisms. For instance, $inc(e, x) = e \xrightarrow{x+1} Id$ and $inc(1) = 1$ defines the homomorphism which increments the value of the first variable. A second step is introduce induction in the description of the homomorphism. For instance, one may generalize the increment operation to the homomorphism $inc(e_1)$ which increments the value of the given variable e_1. A possible approach is to set $inc(e_1)(e, x) = e \xrightarrow{x+1} Id$ whenever $e = e_1$ and otherwise $inc(e_1)(e, x) = e \xrightarrow{x} inc(e_1)$. Indeed, if the first variable is e_1, then the homomorphism increments the values of the variable, otherwise the homomorphism is inductively applied to the next variables. The following proposition formalizes the notion of inductive homomorphisms.

Proposition 6 (Inductive homomorphism). *Let I be an index set. Let $(d_i)_{i\in I}$ be a family of DDDs. Let $(\tau_i)_{i\in I}$ and $(\pi_{i,j})_{i,j\in I}$ be family of homomorphisms. Assume that $\forall i \in I$, the set $\{j \in I \mid \pi_{i,j} \neq 0\}$ is finite. Then the following recursive definition of mappings $(\Phi_i)_{i\in I}$:*

$$\forall d \in \mathbb{D}, \Phi_i(d) = \begin{cases} 0 & \text{if } d = 0 \\ d_i & \text{if } d = 1 \\ \top & \text{if } d = \top \\ \sum_{x\in \mathrm{Dom}(e), j\in I} \pi_{i,j} \circ \Phi_j(\alpha(x)) + \tau_i(\alpha(x)) & \text{if } d = (e,\alpha) \end{cases}$$

defines a family of homomorphisms called inductive homomorphisms. The symbolic expression $\sum_{j\in I} \pi_{i,j} \circ \Phi_j + \tau_i$ is denoted $\Phi_i(e, x)$.

To define a family of inductive homomorphisms $(\Phi_i)_{i\in I}$, one has just to set the homomorphisms $\Phi_i(e, x)$ and the DDDs $\Phi_i(1)$. The two following examples illustrate the usefulness of these homomorphisms to design new operators on DDD. The first example formalizes the increment operation. The second example is a swap operation between two variables. It gives a good idea of the techniques used to design homomorphisms for some variants of Petri net analysis.

Example 2. This is the formal description of increment operation:

$$\begin{aligned} inc(e_1)(e,x) &= \begin{cases} e \xrightarrow{x+1} Id & \text{if } e = e_1 \\ e \xrightarrow{x} inc(e_1) & \text{otherwise} \end{cases} \\ inc(e_1)(1) &= 1 \end{aligned}$$

Let us now detail the application of *inc* over a simple DDD:

$$\begin{aligned} inc(b)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) &= a \xrightarrow{1} inc(b)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\ &= a \xrightarrow{1} b \xrightarrow{3} Id(c \xrightarrow{3} d \xrightarrow{4} 1) \\ &= a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{3} d \xrightarrow{4} 1 \end{aligned}$$

Example 3. The homomorphism $swap(e_1, e_2)$ swap the values of variables e_1 and e_2. It is designed using three other kinds of homomorphisms: $rename(e_1)$, $down(e_1, x_1)$, $up(e_1, x_1)$. The homomorphism $rename(e_1)$ renames the first variable into e_1; $down(e_1, x_1)$ sets the variable e_1 to x_1 and copies the old assignment of e_1 in the first position; $up(e_1, x_1)$ puts in the second position the assignment $e_1 = x_1$.

$$
\begin{array}{ll}
swap(e_1,e_2)(e,x) & = \begin{cases} rename(e_1) \circ down(e_2,x) & \text{if } e = e_1 \\ rename(e_2) \circ down(e_1,x) & \text{if } e = e_2 \\ e \xrightarrow{x} swap(e_1,e_2) & \text{otherwise} \end{cases} \\
swap(e_1,e_2)(1) & = \top \\
 & \\
rename(e_1)(e,x) & = e_1 \xrightarrow{x} Id \\
rename(e_1)(1) & = \top \\
 & \\
down(e_1,x_1)(e,x) & = \begin{cases} e \xrightarrow{x} e \xrightarrow{x_1} Id & \text{if } e = e_1 \\ up(e,x) \circ down(e_1,x_1) & \text{otherwise} \end{cases} \\
down(e_1,x_1)(1) & = \top \\
 & \\
up(e_1,x_1)(e,x) & = e \xrightarrow{x} e_1 \xrightarrow{x_1} Id \\
up(e_1,x_1)(1) & = \top
\end{array}
$$

Let us now detail the application of *swap* over a simple DDD which enlights the role of the inductive homomorphisms:

$$
\begin{array}{ll}
swap(b,d)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) & = a \xrightarrow{1} swap(b,d)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\
 & = a \xrightarrow{1} rename(b) \circ down(d,2)(c \xrightarrow{3} d \xrightarrow{4} 1) \\
 & = a \xrightarrow{1} rename(b) \circ up(c,3) \circ down(d,2)(d \xrightarrow{4} 1) \\
 & = a \xrightarrow{1} rename(b) \circ up(c,3)(d \xrightarrow{4} d \xrightarrow{2} 1) \\
 & = a \xrightarrow{1} rename(b)(d \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1) \\
 & = a \xrightarrow{1} b \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1
\end{array}
$$

One may remark that $swap(b,e)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) = a \xrightarrow{1} \top$.

1.4 Implementing Data Decision Diagrams

In order to write object oriented programs handling DDDs, a programmer needs a class hierarchy translating the mathematical concepts of DDDs, of set operators, of concatenation, of homomorphisms and of inductive homomorphisms. These concepts are translated in our interface by the definitions of three classes (`DDD`, `Hom` and `InductiveHom`) where all the means to construct and to handle DDDs and homomorphisms are given. Indeed an important goal of our work is to design an easy to use library interface; so, we have used C++ overloaded operators in order to have the most intuitive interface as possible.

From the theoretic point of view, an inductive homomorphism Φ is an homomorphism defined by a DDD $\Phi(1)$ and an homomorphism family $\Phi(e,x)$. Inductive homomorphisms have in common their evaluation method and this leads to the definition of a class that we named `InductiveHom` that contains the inductive homomorphism evaluation method and gives, in term of abstract methods, the components of an inductive homomorphism: $\Phi(1)$ and $\Phi(e,x)$. In

order to build an inductive homomorphism, it suffices to define a derived class of the class `InductiveHom` implementing the abstract methods $\Phi(1)$ and $\Phi(e, x)$.

The implementation of our interface is based on the three following units:

- A DDD management unit: thanks to hash table techniques, it implements the sharing of the memory and guarantees the uniqueness of the tree structure of the DDDs.
- An HOM management unit: it manages data as well as evaluation methods associated to homomorphisms. Again the syntactic uniqueness of a homomorphism is guaranteed by a hash table. We use the notion of derived class to represent the wide range of homomorphism types.
- A computing unit: it provides the evaluation of operations on the DDDs, as well as the computation of the image of a DDD by an homomorphism. In order to accelerate these computations, this unit uses an operation cache that avoids to evaluate twice a same expression during a computation. The use of cached results reduces the complexity of set operations to polynomial time. Since inductive homomorphisms are user-defined, we cannot express their complexity.

2 Data Decision Diagrams for Ordinary Petri Nets and Some of Their Extension

In this section, we show how DDDs can be used as a toolkit for the verification of a large class of Petri nets. At first, we introduce Petri nets with marking-dependent valuation enriched by place capacity. From this definition, usual subclasses are defined from ordinary nets to self modifying ones. In the following of the section, we present the key element of a model checker, i.e. the inductive homomorphism encoding the symbolic transition relation associated to each subclass of net. Finally, we show how DDDs can be used in the context of queuing nets.

2.1 P/T-Nets with Marking-Dependent Cardinality Arcs and Capacity Places

A *P/T-Net* is a tuple $\langle P, T, Pre, Post, Cap\rangle$ where

- P is a finite set of places,
- T is a finite set of transitions (with $P \cap T = \emptyset$),
- Pre and $Post : P \times T \to (\mathbb{N}^P \to \mathbb{N})$ are the marking-dependent pre and post functions labelling the arcs.
- $Cap : P \to \mathbb{N} \cup \{\infty\}$ defines the capacity of each place.

For a transition t, ${}^\bullet t$ (resp. $t^\bullet$) denotes the set of places $\{p \in P \mid Pre(p, t) \neq 0\}$ (resp. $\{p \in P \mid Post(p, t) \neq 0\}$).

A marking m is an element of $\mathbb{N}^P$ satisfying $\forall p \in P, m(p) \leq Cap(p)$. A transition t is enabled in a marking m if for each place p, the two conditions

$Pre(p,t)(m) \leq m(p)$ and $m(p) - Pre(p,t)(m) + Post(p,t)(m) \leq Cap(p)$ hold. The firing of a transition t from a marking m leads to a new marking m' defined by $\forall p \in P, m'(p) = m(p) - Pre(p,t)(m) + Post(p,t)(m)$. As usual, this firing rule is extended to sequence of transitions. Moreover, we denote by $Reach(N, m_0)$ the set of markings of the P/T-Net N reachable from the initial marking m_0.

From this definition of nets, we define the following sub-classes by restricting the functions labelling the arcs as well as the capacities associated to the places. Assume $\alpha^-, \alpha^+ \in \mathbb{N}^{P \times T}$.

A *net without capacity restriction* is a P/T-Net satisfying $\forall p \in P$, $Cap(p) = \infty$.

A *constant net* (or *capacity net*) is a P/T-Net for which all the functions on the arcs are constant (i.e. $\forall p \in P, \forall t \in T, Pre(p,t) = \alpha^-(p,t)$ and $Post(p,t) = \alpha^+(p,t)$).

An *ordinary net* is a constant net without capacity restriction.

An *inhibitor net* is a net without capacity restriction and for which the functions labelling the arcs satisfy $\forall p \in P, \forall t \in T, Pre(p,t) = \alpha^-(p,t) \vee Pre(p,t) = 2m(p)$ and $Post(p,t) = \alpha^+(p,t)$. If a transition t has an input place p such that $Pre(p,t) = 2m(p)$ then if $m(p) \neq 0$ then t is not enabled in m. Such an arc is called an inhibitor arc.

A *reset net* is a net without capacity restriction and for which the arc valuation functions satisfy $\forall p \in P, \forall t \in T, Pre(p,t) = \alpha^-(p,t) \vee Pre(p,t) = m(p)$ and $Post(p,t) = \alpha^+(p,t)$. If a transition t has an input place p such that $Pre(p,t) = m(p)$ and t is enabled in m then if m' is the marking reached by firing t, we have $m'(p) = 0$. Such an arc is called a reset arc.

Assume $\beta^-, \beta^+ \in \{0,1\}^{P \times T \times P}$. A *self modifying net* is a net without capacity restriction and for which the arc valuation functions satisfy : $\forall p \in P, \forall t \in T$,

$$Pre(p,t) = \alpha^-(p,t) + \sum_{r \in P} \beta^-(p,t,r) m(r)$$
$$Post(p,t) = \alpha^+(p,t) + \sum_{r \in P} \beta^+(p,t,r) m(r)$$

This last definition is more restrictive than the one in [9] for which the coefficients β are natural numbers.

2.2 DDD and Ordinary Petri Nets

First, we show how to encode the states of an ordinary net. We use one variable for each place of the system. The domain of place variables is the set of natural numbers. The initial marking for a single place is encoded by:

$$d_p = p \xrightarrow{m_0(p)} 1$$

For a given total order on the places of the net, the DDD encoding the initial marking is the concatenation of DDDs $d_{p_1} \cdots d_{p_n}$.

The symbolic transition relation is defined arc by arc. There exists alternative definitions and the one we chose is not the most efficient. Indeed, it induces that the DDD representing a set of markings must be traversed for each arc adjacent to a same transition. It is clear that the definition of a more complicated homomorphism, taking into account all the input and output places of a transition, would be more efficient. However, the arc by arc definition is modular and well-adapted for the further combination of arcs of different net sub-classes. We adopt it for sake of presentation clarity. Notice that all homomorphisms defined in this section are independent of the chosen order.

Two homomorphisms are defined to deal respectively with the pre (h^-) and post (h^+) conditions. Both are parameterized by the connected place (p) as well as the valuation (v) labelling the arc entering or outing p .

$$h^-(p,v)(e,x) = \begin{cases} e \xrightarrow{x-v} Id & \text{if } e = p \wedge x \geq v \\ 0 & \text{if } e = p \wedge x < v \\ e \xrightarrow{x} h^-(p,v) & \text{otherwise} \end{cases}$$
$$h^-(p,v)(1) = \top$$

$$h^+(p,v)(e,x) = \begin{cases} e \xrightarrow{x+v} Id & \text{if } e = p \\ e \xrightarrow{x} h^+(p,v) & \text{otherwise} \end{cases}$$
$$h^+(p,v)(1) = \top$$

The symbolic relation of a given transition t is given by the definition:

$$h_{Trans}(t) = \bigcirc_{p \in t^\bullet} h^+(p, \alpha^+(p,t)) \circ \bigcirc_{p \in {}^\bullet t} h^-(p, \alpha^-(p,t))$$

where the positive coefficients α^+ and α^- are the ones of the Section 2.1. It is important to note that the state coding together with the homomorphisms h_{Trans} ensure that the produced DDDs are well-defined. This algorithm has been evaluated on a set of examples taken from [2].

The table 1 presents the obtained results (on a PowerBook G4, 667Mhz with 512Mo and running MacOs X). For each model is given the value of the parameter, the cardinality of the reachable marking set, the size of the corresponding DDD, the size of the corresponding tree (i.e a DDD without sharing) and the computation time (in second). We can remark that the size of the DDDs remains reasonable for all these results (linear for the dining philosophers) as well as the computation time when the used homomorphisms can be optimized. However, it is clear that this prototype does not compete with the solution presented in [2]. Indeed, in [2], the computation of the reachable marking set for 50 philosophers takes less than 1s.

2.3 Nets with Inhibitor Arcs, Capacity Places and Reset Arcs

The necessary homomorphisms to deal with inhibitor and reset arcs or places with capacities are just adaptations of the previous ones. Indeed, the firing rule of these particular elements implies only local operations. As an example, the following homomorphism is a simple adaptation of h^- to inhibitor arcs.

Table 1. Experimentation results for ordinary nets

	N	reached	DDD size	no sharing	time
philosophers	5	1364	127	11108	0.12
	10	1.86×10^6	267	1.52×10^7	0.68
	50	2.23×10^{31}	1387	1.82×10^{32}	24.48
slotted ring	5	53856	350	507762	5.21
	10	8.29×10^9	1281	8.07×10^{10}	136.37
fms	5	2.89×10^6	225	9.97×10^6	0.76
	10	2.5×10^9	580	8.02×10^9	4.05
	20	6.03×10^{12}	1740	1.87×10^{13}	26.09
kanban	5	2.55×10^6	112	8.98×10^6	2.6
	10	1.01×10^9	257	3.29×10^9	34.83

$$h^i(p)(e,x) = \begin{cases} e \xrightarrow{x} Id & \text{if } e = p \wedge x = 0 \\ 0 & \text{if } e = p \wedge x > 0 \\ e \xrightarrow{x} h^i(p) & \text{otherwise} \end{cases}$$
$$h^i(p)(1) = \top$$

Another adaptation is defined to take into account the capacity of the places:

$$h^c(p,v)(e,x) = \begin{cases} e \xrightarrow{x} Id & \text{if } e = p \wedge x \leq v \\ 0 & \text{if } e = p \wedge x > v \\ e \xrightarrow{x} h^c(p,v) & \text{otherwise} \end{cases}$$
$$h^c(p,v)(1) = \top$$

In a similar way, h^+ can be adapted to deal with reset arcs.

$$h^r(p)(e,x) = \begin{cases} e \xrightarrow{0} Id & \text{if } e = p \\ e \xrightarrow{x} h^r(p) & \text{otherwise} \end{cases}$$
$$h^r(p)(1) = \top$$

The composition of these different homomorphisms allows us to define the symbolic transition relation for nets including inhibitor and reset arcs as well as capacity places. Let t be a transition, its transition relation is:

$$\begin{aligned} h_{Trans}(t) = & \bigcirc_{p \in t^\bullet \wedge Cap(p) \neq \infty} h^c(p, Cap(p)) \\ & \circ \bigcirc_{p \in t^\bullet} h^+(p, \alpha^+(p,t)) \\ & \circ \bigcirc_{p \in {}^\bullet t \wedge Pre(p,t) = \alpha^-(p,t)} h^-(p, \alpha^-(p,t)) \end{aligned}$$

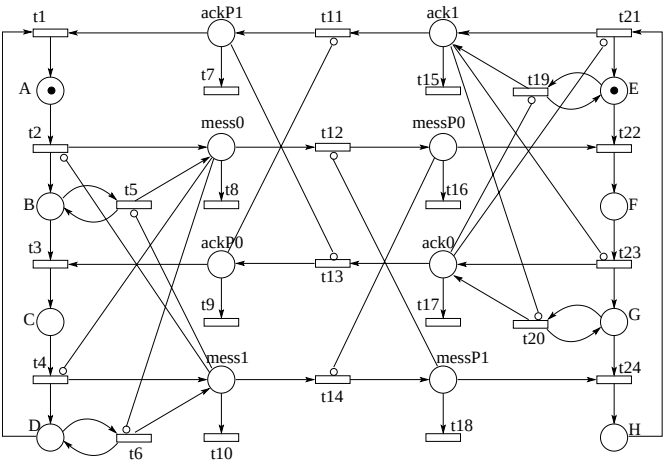


Fig. 2. Inhibitor net (with capacity) of the alternate bit protocol

$$\circ \bigcirc_{p \in {}^\bullet t \wedge Pre(p,t)=2m(p)} h^i(p)$$
$$\circ \bigcirc_{p \in {}^\bullet t \wedge Pre(p,t)=m(p)} h^r(p)$$

The symbolic transition relation given below has been evaluated on the model of Fig.2 originally presented (and explained) in [3]. The table 2 gives the obtained results. The parameter is here the capacities of the places containing the messages and the acknowledgments. We can remark that the number of DDD nodes is constant. However, the number of arcs (not given in the table) is not.

Table 2. Experimentation results for inhibitor, reset and capacity net

	N	reached	DDD size	no sharing	time
alternate bit	5	14688	84	43804	1.89
	10	170368	84	480394	11.21
	20	2.23×10^6	84	6.29×10^6	94.66

2.4 Self Modifying Nets

The design of homomorphisms for self modifying nets is more tricky. Indeed, the expression labelling an arc between two nodes can refer any places of the net and then its evaluation is a global operation.

Of course, the adjacent places of the considered transition can be referenced in the function labelling the arcs. Like in previous subsections, the pre condition evaluation as well as the computation of the reached marking are done in several steps (arc by arc). Let m be the marking before the firing of the considered transition t . In order to properly compute the reached marking, for any place p adjacent to t and referenced in an expression used to fire t, the marking $m(p)$ must be stored. Indeed, this value is necessary for the arc by arc computation. To deal with this specific situation, we duplicate such places in the DDD before doing any other operation. The first occurrence of a place is used to store the current value of the marking during the computation whereas the second one represents the value of the marking before the firing. After all the operations are performed, the duplicates are removed. Duplication and removal are expressed in terms of homomorphisms. Due to their weak interest, these homomorphisms are not presented here.

The evaluation of the pre condition of a transition in a self modifying net is based on the following homomorphisms.

$$\begin{aligned} search(p,i)(e,x) &= \begin{cases} e \xrightarrow{x} e \xrightarrow{x} Id & \text{if } e = p \wedge i = 1 \\ up(e,x) \circ search(p,i-1) & \text{if } e = p \wedge i > 1 \\ up(e,x) \circ search(p,i) & \text{otherwise} \end{cases} \\ search(p,i)(1) &= \top \end{aligned}$$

search is a specialization of the homomorphism *down* presented in the Section 1.3. It searches for the value of the i^{th} occurrence of a given variable. Indeed, in many cases the value to be subtracted is the one associated to the second occurrence of the variable (the original value of the place marking).

$$\begin{aligned} assign^{-}(p,v)(e,x) &= \begin{cases} p \xrightarrow{v-x} Id & \text{if } x \leq v \\ 0 & \text{otherwise} \end{cases} \\ assign^{-}(v)(1) &= \top \end{aligned}$$

$assign^{-}$ is applied on the variable corresponding to the current value of the marking. Knowing the effective value x of the arc valuation (retrieved by *search*) and the current value v of the marking, it evaluates the pre condition and construct the new marking if the latter is satisfied.

$$\begin{aligned} setCst^{-}(p,v)(e,x) &= \begin{cases} e \xrightarrow{x-v} Id & \text{if } e = p \wedge v \leq x \\ 0 & \text{if } e = p \wedge v > x \\ e \xrightarrow{x} setCst^{-}(p,v) & \text{otherwise} \end{cases} \\ setCst^{-}(p,v)(1) &= \top \end{aligned}$$

$setCst^{-}$ is used in the opposite case. The effective value v of the arc valuation is known *a priori*. When the current value of the marking of the place p is found (the first occurrence of the variable), the pre condition is evaluated and the marking modified in case of success.

$$set^-(p,p',i)(e,x) = \begin{cases} assign^-(p,x) \circ search(p',2) & \text{if } e = p \wedge p \neq p' \\ assign^-(p,x) \circ search(p',1) & \text{if } e = p \wedge p = p' \\ p' \xrightarrow{x} setCst^-(p,x) & \text{if } e = p' \wedge p \neq p' \wedge i = 1 \\ p' \xrightarrow{x} set^-(p,p',i-1) & \text{if } e = p' \wedge p \neq p' \wedge i > 1 \\ e \xrightarrow{x} set^-(p,p',i) & \text{otherwise} \end{cases}$$
$$set^-(p,p',i)(1) = \top$$

set^- subtracts to the current value of the place p (its first occurrence), the original value of the place p' (its second occurrence). This assignment is performed according to the order between p and p'. The case when p precedes p' is treated by the first statement of the previous definition by combining a search followed by an assignment. The special situation of a reset arc is taken into account by the second statement of the definition. The search is then limited to the next occurrence of the variable. When p' precedes p, the effective value of the arc valuation is then known before visiting p. The value to be subtracted to the current value of p is the value associated to the second occurrence of p'. Then, an index i is used to count how many occurrences have been already visited. As a consequence, we define $h^{sm^-}(p,p')$ as $set^-(p,p',2)$.

On the other hand, a set of homomorphisms ($assign^+$, $setCst^+$ and set^+) is defined to deal with the post condition. These homomorphisms are symmetric to those presented above. The main difference is there is no condition $x \leq v$ (resp. $v \leq x$) in $assign^+$ (resp. $setCst^-$) and a sum $v + x$ is produced in these homomorphisms. We define $h^{sm^+}(p,p')$ as $set^+(p,p',2)$

The symbolic relation for nets containing self modifying arcs is given below. This transition relation can be easily combined with the one of Section 2.3 to deal with a more general class of net. This combined transition relation has been implemented in our prototype.

$$h_{Trans}(t) = \bigcirc_{p \in t^\bullet}(h^+(p,\alpha^+(p,t)) \circ \bigcirc_{r \in P \wedge \beta^+(p,t,r)=1} h^{sm^+}(p,r)) \\ \circ \bigcirc_{p \in {}^\bullet t}(h^-(p,\alpha^-(p,t)) \circ \bigcirc_{r \in P \wedge \beta^-(p,t,r)=1} h^{sm^-}(p,r))$$

The previous transition relation has been evaluated on the example of Fig. 3 originally presented in [9]. The parameters are the numbers of processes and of

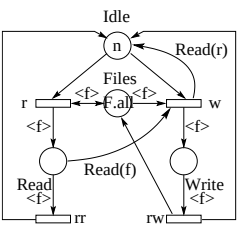


Fig. 3. Self modifying net of readers and preemptive writers

files. The results are presented in Tab. 3. We remark that the size of the DDD is proportionally linear to the value of the parameters. In the case of finite systems (i.e. nets with a finite number of reachable markings), it is well-known that self modifying nets have the same expressive power as ordinary nets. The net of Fig. 3 can then be imitated by an equivalent ordinary net where the transition w is duplicated to take into account the different situations (the different possible number of readers of a given file). Notice that some complementary places must be introduced to test the strict equality and then that the place bounds must be known *a priori*. The last column of Tab. 3 indicates the time needed to treat the equivalent ordinary net. We remark that the concision of the self modifying net allows to take advantage on this alternative.

Table 3. Experimentation results for self modifying net

	N	reached	DDD size	no sharing	time	ord. time
preemptive writers	5×5	873	120	6440	0.62	0.72
	10×10	1.94×10^6	485	1.46×10^7	18.35	24.35
	15×15	4.95×10^9	1100	3.77×10^{10}	137.5	211.9

2.5 Queuing Nets

In this section, we present how DDD can be used to verify ordinary net enriched by lossy queues. In our model, a loss can occur at any position in the queue. Notice that this model cannot be simulated by an ordinary net without introducing intermediary markings.

A *queuing net* is a tuple $\langle \Sigma, P, Q, Q_{loss}, T, Pre_P, Post_P, Pre_Q, Post_Q \rangle$ where:

- Σ is a finite alphabet,
- $\langle P, T, Pre_P, Post_P \rangle$ forms an ordinary net,
- Q is a finite set of queues,
- $Q_{loss} \subseteq Q$ is the set of lossy queues,
- Pre_Q and $Post_Q : Q \times T \to \Sigma \cup \{\epsilon\}$ are the pre and post conditions of the queues.

For a transition t, we denote by ${}^{\triangleright}t$ (resp. $t^{\triangleright}$), the set of queues $\{q \in Q \mid Pre_Q(q,t) \neq \epsilon\}$ (resp. $\{q \in Q \mid Post_Q(q,t) \neq \epsilon\}$).

A marking m is an element of $\mathbb{N}^P \times (\Sigma^*)^Q$. A transition t is enabled in a marking m if for each place p, the condition $Pre_P(p,t) \leq m(p)$ holds and if for each queue q, there exists a word $\omega \in \Sigma^*$ such that $m(q) = Pre_Q(q,t) \cdot \omega$. The firing of t from m leads to a set of new markings constructed from the marking m' defined by $\forall p \in P, m'(p) = m(p) - Pre_P(p,t) + Post_P(p,t)$ and $\forall q \in Q$, if $m(q) = Pre_Q(q,t) \cdot \omega$ then $m'(q) = \omega \cdot Post_Q(q,t)$. A marking reached

from m by firing t is any marking obtained by erasing any number of letters of the words $m'(q)$ for the lossy queue $q \in Q_{loss}$.

The encoding of the states of a queuing net is obtained by generalizing the one used for P/T-nets. We use one variable for each place and each queue of the system. The domain of place variables is the set of natural numbers, while the domain of queue variables is $\Sigma \cup \{\#\}$, the set of messages that may contain a queue and a terminal character "#" (we assume that $\# \notin \Sigma$). The initial marking for a single place and a single queue is encoded by:

$$d_r = \begin{cases} r \xrightarrow{x} 1 & \text{if } r \in P, m_0(r) = x \\ r \xrightarrow{a_0} r \xrightarrow{a_1} r \cdots \xrightarrow{a_n} r \xrightarrow{\#} 1 & \text{if } r \in Q, m_0(r) = a_0 \cdot a_1 \cdots a_n \end{cases}$$

For a given total order $r_1, r_2 \cdots r_n$ on $P \cup Q$, the DDD encoding the initial marking is the concatenation of DDDs $d_{r_1} \cdots d_{r_n}$.

We are now in position to define the homomorphisms used for the arcs related to the queues.

$$h^{q^-}(q,v)(e,x) = \begin{cases} Id & \text{if } e = q \wedge x = v \\ 0 & \text{if } e = q \wedge x \neq v \\ e \xrightarrow{x} h^{q^-}(q,v) & \text{otherwise} \end{cases}$$
$$h^{q^-}(q,v)(1) = \top$$

h^{q^-} tests that the first occurrence of the variable q is associated with the value v and in this case, it removes this occurrence.

$$h^{q^+}(q,v)(e,x) = \begin{cases} e \xrightarrow{v} e \xrightarrow{\sharp} Id & \text{if } e = q \wedge x = \sharp \\ e \xrightarrow{x} h^{q^+}(q,v) & \text{otherwise} \end{cases}$$
$$h^{q^+}(q,v)(1) = \top$$

h^{q^+} searches the last occurrence of the variable q (the one associated with the value $\sharp$) and then introduces the value v before the terminal character at the end of the queue.

$$h^l(q)(e,x) = \begin{cases} h^l(q) + e \xrightarrow{x} h^l(q) & \text{if } e = q \wedge x \neq \sharp \\ e \xrightarrow{x} Id & \text{if } e = q \wedge x = \sharp \\ e \xrightarrow{x} h^l(q) & \text{otherwise} \end{cases}$$
$$h^l(q)(1) = \top$$

h^l is defined to deal with lossy queues. Its role is to produce all the markings obtained by erasing any combination of letters in the word associated to the queue q.

$$\begin{aligned} h_{Trans}(t) = & \bigcirc_{q \in Q_{lost}} h^l(q) \\ & \circ \bigcirc_{q \in t^\triangleright} h^{q^+}(q, Post_Q(q,t)) \circ \bigcirc_{p \in t^\bullet} h^+(p, \alpha^+(p,t)) \\ & \circ \bigcirc_{q \in {}^\triangleright t} h^{q^-}(q, Pre_Q(q,t)) \circ \bigcirc_{p \in {}^\bullet t} h^-(p, \alpha^-(p,t)) \end{aligned}$$

The symbolic relation of a given transition t is given below. Notice that after the firing of a transition, all the markings reached by loosing some messages are produced by applying h^l. In particular, the homomorphism $\bigcirc_{q \in Q_{lost}} h^l(q)$ must be applied to the initial marking.

The previous transition relation has been evaluated on a model of the alternate bit protocol with bounded lossy queues. In our prototype, we have adapted the homomorphism h^c to deal directly with the queue capacities. The experimental results are presented in Tab. 4 and the parameter is the capacity of the queues. Here again, the size of the DDDs is linearly proportional to the value of N.

Table 4. Experimentation results for queuing net

	N	reached	DDD size	no sharing	time
alternate bit	5	630	104	2008	0.21
	10	3355	164	10303	0.53
	20	21105	284	63943	1.57
	50	280755	644	845263	8.46
	100	2.12×10^6	1244	6.37×10^6	33.4

3 Concluding Remarks

DDDs provide a sensible alternative to other decision diagrams and the absence of hypotheses on the variables and their domains gives an important flexibility for the coding of the states (allowing the representation of dynamic data structures or the taking into account of data without *a priori* knowing their domains). The inductive homomorphisms permits the definition of a large class of operators that are specific to a given application domain. While this expressiveness comes necessarily at a cost, our experience shows that intelligent use of advanced techniques such as maximal sharing makes it possible to provide reasonable performance. Moreover, inductive homomorphisms could be adapted to other decision diagrams.

The application to the analysis of different classes of Petri nets presented in this paper has demonstrated the expressiveness of inductive homomorphisms. Indeed, most of the development cost has been spent for the DDD library while the Petri net analyser has been coded in less than one day. In the same way, we may extend this analyser to provide a complete model checker as a CTL checker.

However, the performance offered by this prototype are not completely satisfactory even if many ways of optimization are possible. At first, DDDs are very sensitive to the variable ordering (like other decision diagrams) and no

reordering techniques have been implemented yet. On the other hand, we can remark that many of the operations are local to a variable. Ciardo et al. in [7,2] have proposed a hierarchical structure to reach directly the block of data where the affected variable is located. DDDs can take advantage of such accelerators. Finally, the symbolic analysis of homomorphism compositions used for a given application and their reordering can also be a way to optimize their evaluation.

In the context of a semi-industrial project, the expressiveness of inductive homomorphisms has been put to the test. We have developed a symbolic model checker for circuits modelled by VHDL programs. A large subset of the language, including discrete timeouts, is covered. These experiments encourage us to integrate temporal aspects in our Petri net model.

References

1. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986. 101
2. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. of ICATPN'2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 103–122. Springer Verlag, 2000. 101, 102, 112, 120
3. J. M. Couvreur and E. Paviot-Adet. New structural invariants for Petri nets analysis. In *Proc. of ICATPN'94*, volume 815 of *Lecture Notes in Computer Science*, pages 199–218. Springer Verlag, 1994. 114
4. H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions of Computer-Aided Design*, 18(7), July 1999. 101
5. T. Kolks, B. Lin, and H. De Man. Sizing and verification of communication buffers for communicating processes. In *Proc. of IEEE International Conference on Computer-Aided Design*, volume 1825, pages 660–664, Santa Clara, USA, November 1993. 101
6. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In L. J.M Claesen, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference, DAC'90*, pages 52–57, June 1990. 101
7. A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. of ICATPN'99*, volume 1639 of *Lecture Notes in Computer Science*, pages 6–25. Springer Verlag, 1999. 101, 102, 120
8. E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *Proc. of ICATPN'94*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer Verlag, 1994. 101
9. R. Valk. Bridging the gap between place- and floyd-invariants with applications to preemptive scheduling. In *Proc. of ICATPN'93*, volume 691 of *Lecture Notes in Computer Science*, pages 432–452. Springer Verlag, 1993. 111, 116

Non-controllable Choice Robustness Expressing the Controllability of Workflow Processes

Juliane Dehnert

Technical University Berlin
dehnert@cs.tu-berlin.de

Abstract. Workflow systems are reactive systems. They run in parallel with their environment. They respond to external events and produce events which again have certain effects in the environment. Most approaches in modeling workflow systems assume reasonable behavior of the environment. They disregard malicious requests such as e.g. denial-of-service attacks or hacker attacks trying to misuse the provided services. Hence they do not provide the modeler with a means to check whether their processes react robustly to any possible request from the environment. In this paper we will propose a means to overcome this deficiency. Based on the modeling with Workflow nets we will introduce a new correctness criterion, called *non-controllable choice robustness*. This criterion depicts the ability of a workflow system to react to possible requests from the environment and still guarantee some desired objective. For the definition and the algorithmical installment of *non-controllable choice robustness* parallels to Game Theory are drawn.

1 Introduction

Workflow systems are reactive systems. They run in parallel with their environment. They respond to external events and produce events which again have certain effects in the environment. Most approaches in modeling workflow processes assume reasonable behavior of the environment. Hence they do not provide the modeler with a means to check whether their processes react robustly to any possible request from the environment. These approaches disregard malicious requests such as e.g. denial-of-service attacks or hacker attacks trying to misuse the provided services. In order to inhibit a breakdown of the workflow system and to guarantee smooth processing it is necessary to focus on the possible interactions with the environment already at design time. The modeling technique should provide concepts to describe interactions explicitly. On this basis a correctness criterion should be provided expressing robust behavior of a workflow system against all possible requests coming from the environment.

First approaches that introduce concepts to reflect the interaction of workflow processes with their environment are [3,13] and [11]. The approaches proposed in [3] and [13] are both based on Workflow nets. The interaction with the environment is either modeled via transitions [3] or via communication channels [13].

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 121–141, 2002.
© Springer-Verlag Berlin Heidelberg 2002

In [13] it is planned to provide a criterion stating whether the modeled process is reasonable with respect to the communication. The approach does not assume any possible environment but checks whether there exists one such that the service can be executed properly. The approach described in [3] focuses more on the comparison of different processes with respect to their outwards observable behavior. In [11] a different modeling technique, namely *activity diagrams* from the UML are used to model workflow processes. The authors prefer this technique arguing that their semantics provide a more adequate way to model reactive behavior. They tailor a formal semantics for activity diagrams faithfully fitting the requirements of the application domain. It will be seen in future publications how this tailored semantics can be used to provide powerful modeling support.

The approach introduced in this paper is based on the modeling with Workflow nets. Workflow nets as introduced in [1] have become a popular choice in this application domain. They can express the desirable routing constructs [4] and combine an intuitive graphical representation with an excellent formal foundation [2]. Furthermore they are supported by a rich variety of existing analysis techniques and tools. In this paper we will enhance the modeling technique slighty including the interactive behavior. In order to do so we do not add further concepts but differentiate between transitions that reflect internal and external behavior. On the base of this differentiation we provide a means to check whether a workflow process may react robust to whatever comes from the environment. We introduce a new correctness criterion called *non-controllable choice robustness* and provide an algorithm to verify it. For its definition we refer to the rich theory on Controller Synthesis using terms and notions from Game Theory.

The results presented in this paper extend the quality measure for workflow models providing the application developer with a further means to check the reasonability of their workflow model. The proposed approach will not defend from denial-of-service attacks but will provide a first means to improve the robustness of workflow systems against malicious interactions from outside.

The remainder of the paper is organized as follows: First (Section 2), we will introduce Workflow nets and summarize some related results from Petri net theory - especially recapitulate recent correctness criteria as soundness and relaxed soundness. In Section 3 we describe the requirements that come along with the new perspective and introduce their modeling. Subsequently, we exemplarily show deficiencies of the recent correctness criteria and introduce the new criterion. In Section 4 we propose an algorithm checking the property, and verify its correctness. At the end of the paper (Section 5) we point to the relationship between the three criteria, present some conclusions and outline future work.

2 Workflow Nets

In this section the used modeling technique is introduced and some related results from Petri net theory are summarized.

2.1 Preliminaries

In this paper Petri nets are used for the modeling of workflow processes. The suitability of Petri nets for this application domain has been examined and discussed extensively in the literature (e.g. [2,5]). Van der Aalst [1] applies Petri net theory to workflow specification and introduces Workflow nets (WF nets). A WF net is a Place/Transition net, which has a unique source place (i) and a unique sink place (o). A token in the source place (i) corresponds to a case which needs to be handled, a token in the sink place (o) corresponds to a case that has been handled. The process state is defined by a marking. In addition, a WF net requires all nodes (i.e. transitions and places) to be on some path from i to o. This ensures that every task (transition) and every condition (place) contributes to the processing of cases. The formal description for Petri net (Place/Transition net), WF net and related results are given below:

Definition 1 (Petri net). *A Petri net is a triple* (P, T, F)*:*
P *is a finite set of places,*
T *is a finite set of transitions* $(P \cap T = \emptyset)$*,*
$F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation).*

$a\bullet = \{b | (a, b) \in F\}$ is called post-set of a.
$\bullet a = \{b | (b, a) \in F\}$ is called preset of a

The global state of a Petri net, called a *marking*, is the distribution of tokens over places $m : P \longrightarrow N$. A marking m changes by firing a transition t. A transition t may fire only if it is enabled.

A transition $t \in T$ is said to be enabled by the marking m iff $\forall p \in \bullet t, p \in P : m(p) \geq 1$. In this case t can fire. Firing of transition t leads from marking m to marking m', where

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

written as $m \xrightarrow{t} m'$. A finite sequence $\sigma = t_1 t_2 t_3 ... t_{n-1}$ of transitions is called a *finite firing sequence, enabled at* m_1 if there exist markings $m_2, ...m_n$, such that $m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} ... \xrightarrow{t_{n-1}} m_n$. We write $m_1 \xrightarrow{\sigma} m_n$. The empty sequence ϵ is enabled at any marking m and satisfies $m \xrightarrow{\epsilon} m$. A marking m_n is called reachable from m_1 (notation $m_1 \xrightarrow{*} m_n$) iff there is a finite firing sequence σ such that $m_1 \xrightarrow{\sigma} m_n$. An infinite sequence $t_1 t_2 t_3 ...$ is called *infinite firing sequence, enabled at* m_1, if there exist markings $m_2, m_3 ...$, such that $m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} m_3 \xrightarrow{t_3} ....$

A pair (PN, m_i) is called a system and denotes a Petri Net PN with an initial marking m_i.

The set of firing sequences of a system can be embedded into a graph. Every firing sequence corresponds to some path in that graph and vice versa.

Definition 2 (Reachability graph). *Let $S = (PN, m_i)$ be a system. Let V be the set of reachable markings in (PN, m_i). Let $E \subseteq (V \times T \times V)$ be a set of T-labeled edges, $E = \{(m, t, m') | m \xrightarrow{t} m'\}$ then $G = (V, E)$ is called reachability graph (RG) of (PN, m_i).*

A path π of the reachability graph is a (finite or infinite) sequence of labeled edges

$$\pi = (m_i, l_1, m_1)(m_1, l_2, m_2)(m_2, l_3, m_3), ...$$

Through the edges a predecessor/successor relationship is defined on the elements of V. m_1 (m_n) is predecessor (successor) of m_n (m_1) if there exist a path from m_1 to m_n : $(m_1, l_1, m_2)...(m_{n-1}, l_{n-1}, m_n)$. We call m (m') immediate predecessor (successor) of m' (m) if (m, t, m') is element of $RG.E$. The functions $pred, succ, Pred, Succ : V \longrightarrow 2^V$ denotes the set of (immediate) predecessors, respectively successors for a state v. So $Pred_{RG}(o)$ denotes the set of states $m \in V$: exists a path π in RG leading from m to o: $\pi = (m, l, m_1)(m_1, l_1, m_2)....(m_n, l_n, o)$.

The construction of the reachability graph RG is straightforward, although termination cannot be guaranteed, because it might be infinite. The graph is infinite if the corresponding Petri net is unbounded.

Definition 3 (bounded, safe). *A system (PN, m_i) is bounded iff for each place p there is a natural number n such that for every reachable marking the number of tokens in p is less than n. The net is safe iff for each place the maximum number of tokens does not exceed 1.*

Definition 4 (strongly connected). *A Petri net is strongly connected iff for every pair of nodes (i.e. places and transitions) x and y, there is a path leading from x to y.*

A WF net is a special Petri net, defined as follows:

Definition 5 (WF net). *A Petri net $PN = (P, T, F)$ is a WF net, iff:*
(i) PN has two special places: i and o. Place i is a source place: $\bullet i = \emptyset$ and place o is a sink place: $o\bullet = \emptyset$.
(ii) If we add a transition t^* to PN which connects place o with i, then the resulting Petri net $\overline{PN}$ is strongly connected.

An example for a process specified by a WF net is shown in Figure 1. It describes the arrangement for planning a trip. It consists of parallel booking activities for flights and hotels.

In the following we will introduce the properties soundness and relaxed soundness as means to check correctness of a process specified by a WF net.

2.2 Correctness Criteria for WF Nets

Van der Aalst [2] introduced soundness as a correctness criterion for WF nets. Soundness ensures that, the process can *always* terminate with a single token in

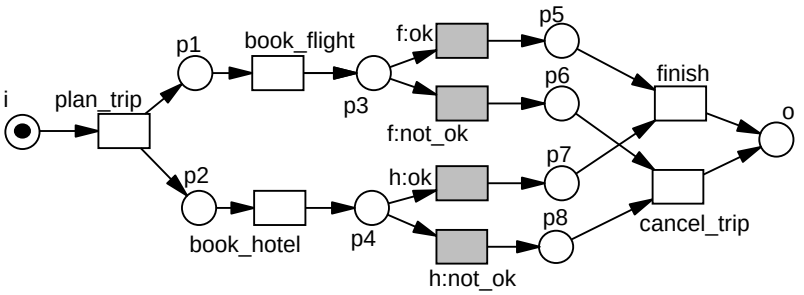


Fig. 1. Example (a): Book Flight and Hotel

place o and all the other places are empty. In addition, it requires that there is no dead task, i.e. each existing task can be executed. Next we show soundness as defined in [2]. Note that we use i and o as identifier for the start- and end places as well as to depict the markings where these places contain a token. The respective meaning can be gathered from the context.

Definition 6 (Soundness).
A process specified by a system $S = (PN, i)$ is sound iff:
(i) For every state m reachable from state i, there exists a firing sequence leading from state m to state o. Formally: $\forall_m (i \xrightarrow{*} m) \Rightarrow (m \xrightarrow{*} o)$.
(ii) State o is the only state reachable from state i with at least one token in place o (proper termination). Formally: $\forall_m (i \xrightarrow{*} m \wedge m \geq o) \Rightarrow (m = o)$
(iii) There are no dead transitions in S.
Formally: $\forall_{t \in T} \exists_{m,m'} (i \xrightarrow{*} m \xrightarrow{t} m')$

In previous publications ([9,8,7]) we argued that soundness is sometimes not adequate but too strict. We therefore proposed the use of a weaker soundness version: *relaxed soundness* as introduced in [9]. The *relaxed soundness* criterion only ensures that there *exist* a reasonable number of executions which terminate properly.

We will define the term *sound firing sequence* to explain the differences between the criteria soundness and relaxed soundness formally.

Definition 7 (Sound Firing Sequence).
Let $S = (PN, i)$ be a system. A firing sequence $t_1 t_2 ... t_k$ is called sound iff there exist transitions $t_l ... t_n$ and markings $m_1, \ldots, m_{n-1}$ such that
$(i \xrightarrow{t_1} m_1 \ldots \xrightarrow{t_k} m_k \xrightarrow{t_l} m_l ... \xrightarrow{t_{n-1}} m_{n-1} \xrightarrow{t_n} o)$.

A sound firing sequence can be extended such that state o is reached. Note that in a sound WF net *all* firing sequences are sound. Relaxed soundness only requires that there *exist* a reasonable number of sound firing sequences. In fact we require so many sound firing sequences that each transition is contained in one of them. Note that soundness implies relaxed soundness.

Definition 8 (Relaxed Soundness).
A process specified by a system $S = (PN, i)$ *is relaxed sound, iff every transition is element of a sound firing sequence.* $\forall t \in T \exists m, m' : (i \xrightarrow{*} m \xrightarrow{t} m' \xrightarrow{*} o)$.

Intuitively relaxed soundness means that there exist enough executions which terminate properly. Enough means at least so many that every transition is covered. So a relaxed sound WF net may have firing sequences which do not terminate properly, but deadlock before or leave tokens in the net. In spite of that relaxed soundness is still reasonable because it requires that all intended behavior has been described correctly.

The process shown in Figure 1 is relaxed sound. There exist sound firing sequences containing all transitions:

Example (a): Book Flight and Hotel:
$plan_trip, book_flight, book_hotel, h{:}ok, f{:}ok, finish$ and
$plan_trip, book_flight, book_hotel, f{:}not_ok, h{:}not_ok, cancel_trip.$

The process is not sound, because there exist firing sequences which do not terminate properly. The process deadlocks if only one of the bookings succeeds but the other one fails.

3 Robust Interaction with the Environment

As mentioned in the introduction the objective of this paper is to provide a workflow property that describes the ability to control the workflow execution independently from the behavior of the environment. After introducing some relevant concepts with respect to the interaction between a workflow and its environment, we motivate the need for such a property by means of some examples. We then define the property *non-controllable choice robustness* and apply it to the examples.

3.1 Reflect the Interaction with the Environment

Workflow systems are reactive systems. They run in parallel with their environment and respond to incoming external events. An external event could be an incoming query, an acknowledgment from a customer, a message from another company, an information from a cooperating partner or just a timeout. Special tasks within the process are linked to the occurrence of external events reflecting the behavior of the environment.

In general we assume a workflow controller to schedule the tasks within a workflow system. It decides which enabled tasks are executed and when. The workflow controller can not force an external event to occur. The kind of external event as well as its occurrence time are beyond its scope. Its clear that the workflow control cannot e.g. force a reader to return her books to the library by now. Thus tasks that are triggered externally, e.g. receiving the books, are therefore also beyond the scope of the local workflow control. These tasks are

called non-controllable. Their execution can not be forced by the local workflow control, but only depend on the kind of incoming event.

A second group of non-controllable task are tasks that depend on the evaluation of external data. It is out of the scope of the local workflow control whether a customer is credit worthy or not, but depending on the outcome of a corresponding check different tasks will become executed.

To refer to these differences we will split the set of transitions T of a WF net $PN = (P, T, F)$ into disjoint sets of *controllable* and *non-controllable* transitions: $T = T_C \cup T_{NC}$. *Controllable* transitions model tasks whose executions are, in contrast to the execution of *non-controllable* tasks, covered by the local workflow control. *Controllable* transitions have been denoted by white boxes whereas *non-controllable* transitions are represented by filled rectangles.

Non-controllable tasks within workflow systems are often part of a choice. They become supplemented by an alternative task triggered through a timeout[1]) which returns the control to the local workflow in case none of the expected external events occur.

Correspondingly we consider non-controllable transition always to be part of a choice[2] and refer to such choices as *non-controllable choices*. In reverse we speak about *controllable choices* if the choice only contains controllable transitions.

Non-controllable choices are required to be free choice. This property guarantees that the corresponding tasks only depend on a specific interaction with the environment and are not linked to some further condition.

Definition 9 (Non-Controllable Choice). *A* non-controllable choice *is a cluster C of only non-controllable transitions: $C \subseteq T_{NC}$. The contained transitions are free-choice: $\forall t \in C : \forall s \in {\bullet t} : s{\bullet} = C$ (cf. [10]).*

Note that we consider all non-controllable transitions to be part of a *non-controllable choice*: $\bigcup_{C_i} = T_{NC}$.

Non-controllable choices depict choices whose outcome depends on interaction with the environment. An orthogonal criterion to distinguish choices is the moment of choice, thus the moment one of the alternative tasks is executed. In this context it is distinguished between implicit and explicit choice (cf. [2,4]). An implicit choice (also deferred choice) is made the moment an external trigger event occurs. An explicit choice is made the moment the task is completed. In this paper we will abstract from the moment the choice is initiated but focus on the outcome of a choice.

The choice in Figure 2(i) is part of the library process "Return Books". Upon borrowing books the system waits for the reader to bring the books back or to ask for an extension. If the reader does neither return the books in time nor extends the validity, a reminder is sent. It is a non-controllable choice because the decision for a certain task to be executed, only rests upon the kind of external event (reader, notification, timeout).

[1] Tasks that are triggerred through a timeout are also considered non-controllable.

[2] This is no restriction because choices containing only one transition are also allowed.

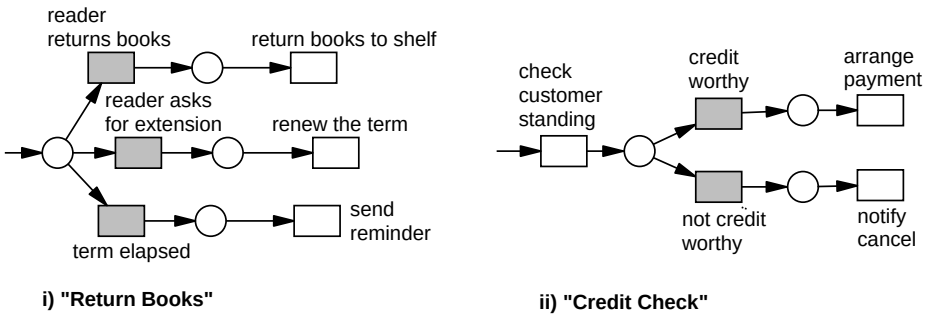


Fig. 2. Examples for non-controllable choices

Figure 2(ii) gives an example for a choice whose outcome depends on the evaluation of external data. The choice is part of the process "Handling Order". The credit-worthiness of a customer is checked. It is decided either *credit worthy* or *not credit worthy*. Depending on the result an according task (*arrange_payment* or *notify_cancel*) is executed.

3.2 Ability to Control a Process

It is our objective to provide a criterion which describes the possibility to control a process independently from the behavior of the environment. It is clear that non-controllable choices have to be considered in defining such a criterion because non-controllable transitions cannot be influenced by a controller. A process can be controlled if there exists a way to terminate properly, independently from non-controllable choices. A bad combination of non-controllable, or controllable and non-controllable choices, may inhibit proper termination. This could happen although the process definition is relaxed sound. Relaxed soundness only makes a proposition about the existence of proper executions, but does not necessarily cover all possible combinations of choices. We will discuss two examples which cannot be controlled, guaranteeing proper termination, although they are relaxed sound.

First we look back at the "Planning Trip"-example from Figure 1. It consists of parallel booking activities for flights and hotels. One possibility of termination is modeled via the transition *finish* - here both activities succeeded (*f:ok, h:ok*) such that the trip can be started. The other possibility is modeled via the transition *cancel_trip*. The trip is canceled if neither a hotel (*h:not_ok*) nor a flight (*f:not_ok*) can be found. These decisions are modeled as non-controllable choices because their computation relies on external data.

Although the process is relaxed sound it cannot always be controlled properly. The computation of the bookings are non-controllable. They cannot be influenced by a controller. The system deadlocks if only one of the bookings succeeds but the other one fails.

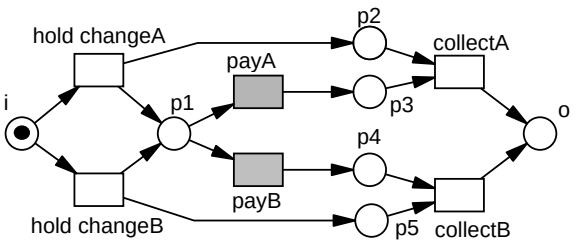


Fig. 3. Example (b): Provide Change

Another example is shown in Figure 3. It describes the work of a cashier. Predicting the behavior of the customer the cashier already holds a predicted amount of change waiting for the customer to pay. This is modeled via the controllable choice *hold changeA* or *hold changeB*. The payment of the customer is modeled via activities *payA* and *payB*. If the cashier had guessed correctly (the amount of change held by the cashier corresponds to the sum provided by the customer) then the money is exchanged. This is modeled via transitions *collectA* or *collectB*. The choice of the customer *payA* and *payB* is non-controllable. The amount the customer chooses cannot be influenced but only depends on the fill level of the customers purse and her mood. The process definition in Figure 3 is relaxed sound. There exist sound firing sequences containing all transitions:

Example (c): Provide Change:
$hold\,changeA, payA, collectA$ and $hold\,changeB, payB, collectB$

Although relaxed sound it cannot be scheduled properly in any case, because the controllable choice *hold changeA* or *hold changeB* lies before the non-controllable choice and cannot react anymore if a wrong amount has been chosen by the customer. In this case the modeled process does not terminate properly but deadlocks.

These process examples show combinations of controllable and non-controllable choices which prevent the possibility to control the execution such that proper termination is guaranteed. Although the processes are relaxed sound it is not possible to force the process to choose between alternative transitions correctly because the non-controllable choices can not be influenced. In case of adverse decision the execution deadlocks.

The given examples illustrate that the property *relaxed soundness* does not cover controllability. It is not possible to guarantee proper termination as soon as requests from outside the system are considered or data dependent decision influences the behavior of the system.

To support the modeler with a notion of correctness we will introduce a further criterion which describes the controllability. We will denote the desired property of a system *non-controllable choice robustness*. If a system is non-controllable choice robust (short: robust), it is possible to control the execution such that it terminates properly.

To define the criterion we will look at our problem from another perspective. Let us think of it as a two-person game between the workflow controller and the environment as its adversary. We regard the environment as an opponent that is trying to interfere in the process execution such that a not-sound firing sequence is generated. The question is whether the workflow controller can win the game that is react to the moves of the adversary and thus terminate properly.

Looking at our setting under this perspective we can refer to profound results of Controller Synthesis based on the use of terms and notions from Game Theory [16,14,6,18]. In the next paragraph we will refer to these notions and deduce results for the application domain of workflow management.

3.3 Game, Play and (Winning-) Strategy

To start with we will recall terms as *game*, *play*, *strategy* and *winning strategy*. The definitions have been adapted from the notions used in [18].

A game is defined as a tuple $(\mathcal{G}, \phi)$ consisting of a game graph $\mathcal{G}$ and a temporal formula ϕ (winning condition).

A *game graph* is of the form $\mathcal{G} = (Q, Q_0, Q_1, E, q_i, F)$ where Q is a finite set of states, Q_0, Q_1 define a partition of Q (depicting the states where it is the turn of player 0 (1) to perform an action) and $E \subseteq (Q_0 \times Q_1) \cup (Q_1 \times Q_0)$ is a set of edges. Note that the underlying graph is required to be bipartite with respect to the state transitions. $q_i \in Q$ is the initial state of the game and $F \subseteq Q$ is a set of accepting states.

A play on a game graph $\mathcal{G}$ corresponds to a path ρ in $\mathcal{G}$ starting in q_i. The decision who wins a play is fixed by the winning condition ϕ. The first player wins a play ρ if ρ satisfies ϕ. Otherwise it looses the play. In this paper we will focus on simple liveness conditions (Büchi-Condition). For a survey of possible winning condition the reader is referred to [18].

A strategy for a given game is a rule that tells a player how to choose between several possible actions in any game position. A strategy is good if the player[3], by following these rules, always wins no matter what the environment does [16]. A strategy[4] for player 0 in the game $(\mathcal{G}, \phi)$ can be depicted as fragment $(SG \subseteq \mathcal{G})$ of the game graph. A strategy SG is a winning strategy for player 0 if all the plays on SG are winning.

A winning strategy SG is called *maximal* iff there does not exists a winning strategy SG' such that $SG \subset SG'$.

3.4 Controller Synthesis for the Application Domain of Workflow Management

In the following we will adapt the introduced notions for the application domain of workflow management. It would be natural to define the game graph on the

[3] All terms are defined from the perspective of player 0 - the workflow controller.

[4] We only consider no-memory strategies, where the next move only depends on the last state cf. [18].

base of the reachability graph RG of a system $S = (PN, i)$, whereas q_i corresponds to i and the set of accepting states F to $\{o\}$. The game graph than is of the form $\mathcal{G}_{RG} = (V_{RG}, E_{RG}, i, o)$.

This adaption is not straightforward, as the set of states in V is not bipartite. This is introduced through the possible concurrent behavior described within the Petri net model. The player (workflow controller and environment) do not have to move in turn and in some states both could make the next move. The moves of the different player are only reflected through the labels at the state transitions. In case the label t of a state transition is a controllable transition $t \in T_C$ then a controller move is represented. Is the state transition labeled with a non-controllable transition $t \in T_{NC}$ it corresponds to a move of the environment. We will make use of this distinction and use (V_{RG}, E_{RG}, i, o), where E_{RG} can be decomposed with respect to the labeling, as game graph. A strategy for the workflow controller can then again be defined as a fragment of the game graph.

Definition 10 (Strategy).
Let $(\mathcal{G}_{RG}, \phi)$ be a game. Let $\mathcal{G}_{\mathcal{RG}} = (V_{RG}, E_{RG}, i, o)$ be the game graph, where $RG = (V_{RG}, E_{RG})$ is the reachability graph of a system $S = (PN, i)$. $SG \subseteq RG$, with $SG = (V_{SG}, E_{SG})$, $V_{SG} \subseteq V_{RG}, E_{SG} \subseteq E_{RG}$, is a strategy if:

1. *$i \in V_{SG}$ and*
2. *for each $v \in V_{SG}$ there is a directed path from i to v in SG.*
3. *SG is self-contained with respect to the possible moves of the adversary: if $t, t' \in T_{NC} : (m, t, m') \in E_{SG}$ and $(m, t', m") \in E_{RG} : (m, t', m") \in E_{SG}$.*

This grasp on the term strategy, as it is usually defined in controller synthesis (e.g. [18]) is not very strong. It enables the possibility that certain choices are never presented to the environment. For our setting a somehow stronger understanding is necessary. A strategy should incorporate the requirement that all possible moves of the environment have to be covered at least once.

Definition 11 (Complete Strategy). *Let $(\mathcal{G}_{RG}, \phi)$ be a game defined on the reachability graph RG of a system $S = (PN, i)$. Let SG be a strategy for the player 0. The strategy SG is called complete if all possible moves of the adversary are covered. Formally: $\forall t \in T_{NC} \exists m, m' \in V_{SG}, (m, t, m') \in E_{SG}$.*

With this means it is now possible to express non-controllable choice robustness of a workflow system.

Definition 12 (Non-Controllable Choice Robustness (short:Robustness)). *Let $(\mathcal{G}_{RG}, \phi)$ be a game defined on the reachability graph RG of a system $S = (PN, i)$. Let ϕ be the conditon of "proper termination" ($\Box\Diamond o$ - in LTL parlance [12]). A process specified by a WF system $S = (PN, i)$ is robust if there exist a complete winning strategy SG for the workflow controller satisfying ϕ.*

Therefore SG additionally meets the following requirements:

1. *$o \in V_{SG}$*
2. *for each $v \in V_{SG}$ there is a directed path from v to o*

A WF system is robust (short: robust) if there exists a subgraph of the reachability graph, which starts in i, ends in o, contains all non-controllable state transitions and has only controllable state transition leading out of the subgraph. The existence of such a subgraph guarantees that it is possible to reach state o (terminate properly) independently from the execution of non-controllable choices. While all non-controllable transitions (all possible moves of the adversary) are covered by the strategy there always exists a way to react and to terminate properly. Hence, if a WF system is robust, the workflow controller can maintain the proper termination against all possible adversary moves.

The process definitions of Figure 1 and 3 are not robust, because the required subgraph does not exist. Figure 4 and 5 show the reachability graphs of both processes. Non-controllable choices have been depicted as bow around the corresponding state transitions.

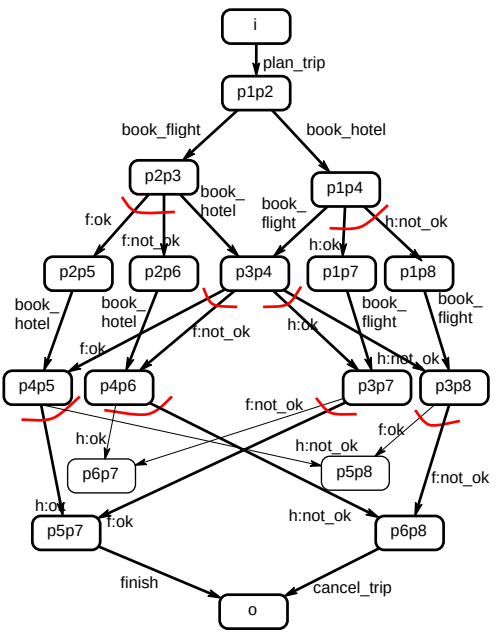


Fig. 4. RG for process (a)

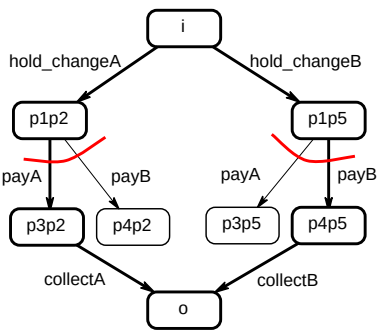


Fig. 5. RG for process (b)

The subgraphs depicted by thickening appendant state transitions satisfy all requirements except 10.3.

In the following we will introduce slight changes to the process definitions to make them robust. Figures 6 and 7 show the modified process definitions. In the "Booking Trip" example process one further activity is introduced to cope with the deadlock situations. The transitions *cancel_trip(a)* and *cancel_trip(b)* are

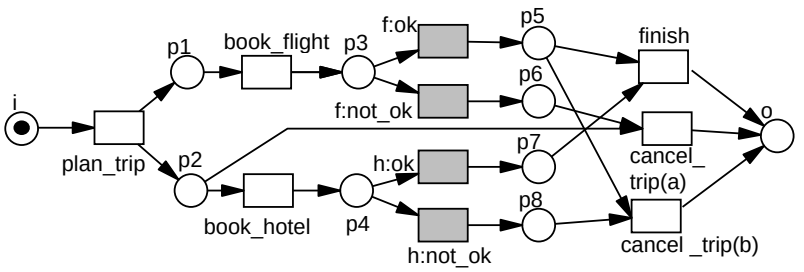


Fig. 6. (a) Book Flight and Hotel II

executed if either no flight could be found (*cancel_trip(a)*) or the flight booking succeeded but no hotel could be booked (*cancel_trip(b)*).

In the process definition of the second example (7) we added two further activities handling the case that the predicted change does not fit to the sum provided by the customer. The transitions *changeBA* and *changeAB* model the behavior of the cashier adapting to the occurred situation by changing the change. Both models are now robust. There exists subgraphs of the reachability graphs, which start in state i, end in state o, cover all non-controllable state transitions and only have controllable state transition leading out. The graphs are shown in Figure 8 and 9.

The existence of a winning strategy for the workflow controller states that it is possible to guarantee proper termination although the non-controllable transitions are beyond its scope.

The first strategy (cf. Figure 8) foresees to delay the execution of the hotel booking until the flight-booking has been finished. Depending on that result the execution proceeds as follows: if the result was negative (*f:not_ok*) the trip is canceled directly (*cancel_trip(b)*) otherwise the hotel booking is initiated.

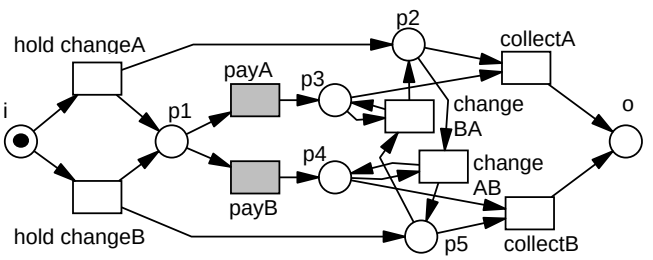


Fig. 7. (b) Provide Change II

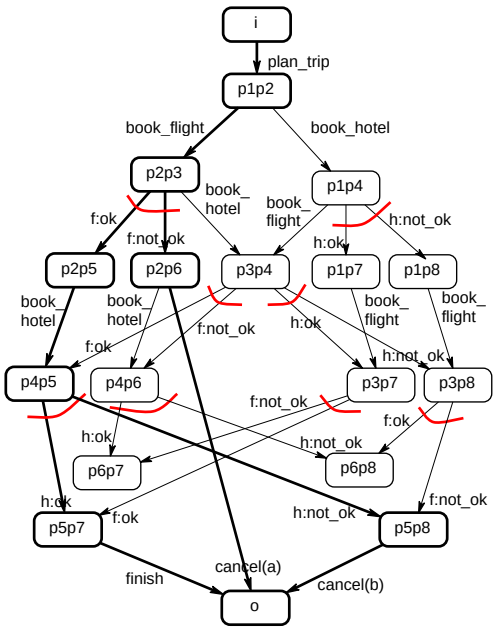


Fig. 8. RG for process (a)II

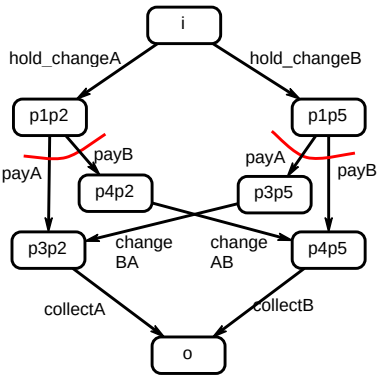


Fig. 9. RG for process (b)II

The second strategy (cf. Figure 7) coincides with the reachability graph. This means that the workflow controller does not has to interfere the process execution because all possible executions terminate properly.

Having found that a particular workflow net has a winning strategy in order to be sure of having a proper termination it seems necessary to restrict the behavior of the workflow to the corresponding subgraph $SG \subseteq RG$.

The WF net should become modified reflecting only the desired behavior. Within future work we hope to release the modeler from that task. We hope to derive rules describing how general synchronization pattern must be introduced into the robust workflow model in order to restrict its behavior to SG. With the help of these rules the modification of the WF net will be generated automatically or at least become supported.

To introduce synchronization pattern means to determine the way the process is executed. Installing a controller for the "Planning Trip"-example (cf. Figure 6) implementing the winning strategy of Figure 8 we would restrict the behavior of the process in a pessimistic manner. By waiting for the outcome of the first non-controllable choice the possibility to execute the activities concurrently gets lost. Following a pessimistic approach the installment of a strategy sequentializes the process executions. In favor of avoiding faulty situations lengthy executions are accepted.

In contrary an optimistic approach would maintain the possibility to execute the activities concurrently but introduce new activities that allow to reset

from a deadlock situation. In favor of supporting parallel execution of depending threads here additional costs introduced through possibly necessary recovery are accepted. The improved WF net from Figure 7 implements an optimistic approach. The recovery behavior is integrated through the new tasks *changeAB* and *changeBA*.

Detaching the last step from the scope of the modeler also enables to delay the decision for a certain strategy. This releases the designer to think about efficiency aspects already at design time and provides a high degree of flexibility with respect to varying priorities. Changing the strategy than does not require to revise the whole model, but only to generate a new controller model.

In the following section we will provide an algorithm that checks whether a given process is robust or not and in the positive case computes a winning strategy for the workflow controller.

4 How to Verify Robustness

There exist various algorithms in the literature that solve the controller synthesis problem in an effective manner, e.g. [15,14,18]. Although it would be possible to transform the reachability graph into a proper game graph and than apply the proposed algorithms we prefer to provide an algorithm that operates on the reachability graph, in order to provide a consistent picture.

Figure 10 shows the algorithm computing the subgraph $SG = (V_{SG}, E_{SG})$. Let $S = (PN, i)$ be a WF system and $T_{NC} \subseteq T$ be the set of non-controllable transitions in S. Let $RG_S = (V_{RG}, E_{RG})$ be the corresponding reachability graph. As prerequisite furthermore the set $Pred_{RG}(o) \subseteq V_{RG}$ of predecessors of state o is computed beforehand. The operation mode of the algorithm is to stepwise color state transitions that belong to the desired subgraph starting from state o. State transition that correspond to controllable transitions are colored immediately. State transitions labeled with non-controllable transitions are colored only if all state transitions corresponding to the same choice lead into states from where state o remains reachable ($Pred_{RG}(o)$). Prerequisite for application of the proposed algorithm is the finiteness of the reachability graph. This means that the algorithm is restricted to check only bounded systems for robustness.

In the next paragraph we will verify the correctness- and completeness of the proposed algorithm.

(i) (Initialization). Set $V_{SG} := \{o\}$; $E_{SG} := \emptyset$; *(* start from state o *)*
(ii) (Iteration). Repeat Until V_{SG} does not change anymore

(a) select $m \in V_{SG}, (m', t, m) \in E_{RG}$
CASE
(b) $t \notin T_{NC} : V_{SG} := V_{SG} \cup \{m'\}$; $E_{SG} := E_{SG} \cup \{(m', t, m)\}$
(color adjacent controllable state transitions *)*

(c) $t \in T_{NC}$: If $succ_{RG}(m') \subseteq Pred_{RG}(o)$ Then
$V_{SG} := V_{SG} \cup \{m'\} \cup succ_{RG}(m')$;
$E_{SG} := E_{SG} \cup \{(m', t, m)\}$;
For $m^* \in succ_{RG}(m')$ Do
select $(m', t', m^*) \in E_{RG}$;
$E_{SG} := E_{SG} \cup \{(m', t', m^*)\}$ Done
(color adjacent non-controllable state transitions, but only if all state transitions of the same choice do not lead out of the area from where state o can be reached *)*

(iii) (Test). If $i \notin V_{SG}$ Then
print(S is not robust); Abort;
(check whether state i has been colored *)*

(iv) (Iteration). For $m \notin Succ_{SG}(i)$ Do
$V_{SG} := V_{SG} \setminus \{m\}$;
For $m' \in succ_{SG}(m)$ Do
select $(m, t, m') \in E_{SG}$;
$E_{SG} := E_{SG} \setminus \{(m, t, m')\}$ Done
Done (* outer loop *)
(clean resulting subgraph, delete elements that are not reachable from i *)*

(v) (Iteration). For $m \in V_{SG}$ Do
If $m \notin Pred_{SG}(o)$;
Then print(S is not robust); Abort;
Done
(check reachability of state o *)*

(vi) (Iteration). For $(m, t, m') \in E_{SG}$ Do
If $t \in T_{NC}$ Then $T_{NC} := T_{NC} \setminus t$; Done
(check which non-controllable transitions are covered *)*

(vii) (Termination). If $T_{NC} = \emptyset$ *(* all non-controllable transitions are covered *)*
Then print(S is robust); output(V_{SG}, E_{SG});
Else print(S is not robust);

Fig. 10. Algorithm: Robustness

4.1 Verification of the Algorithms Correctness and Completeness

We will prove the following proposition:

Proposition 1. *The WF system $S = (PN, i)$ is robust iff the algorithm terminates correctly.*

Remember that a WF system is robust if there exists a complete winning strategy for the workflow controller satisfying the property of proper termination. A strategy $SG \subseteq RG$ is a winning strategy for the workflow controller if the following properties are satisfied:

1. it contains state i: $i \in V_{SG}$ (cf. Def. 10.1),
2. for each $m \in V_{SG}$ there is a directed path from i to m (cf. Def. 10.2),
3. $\forall t, t' in T_{NC}\ (m, t, m') \in E_{SG},\ (m, t', m") \in E_{RG} : (m, t', m") \in E_{SG}$ (cf. Def. 10.3)
4. $\forall\ t \in T_{NC} \exists m, m' \in V_{SG}, (m, t, m') \in E_{SG}$ (cf. Def. 11)
5. $o \in V_{SG}$ (cf. Def. 12.1)
6. for each $v \in V_{SG}$ there is a directed path from v to o (cf. Def. 12.2)

We first prove the direction: If the algorithm terminates correctly, then the WF system is robust. We therefore prove that the subgraph SG produced by the algorithm satisfies the properties mentioned above.

Proof. Let $S = (PN, i)$ be a WF system. Let $RG_S = (V_{RG}, E_{RG})$ be the corresponding reachability graph. Let $SG \subseteq RG_S$ with $V_{SG} \subseteq V_{RG}$ and $E_{SG} \subseteq E_{RG}$ be the result of the algorithm applied to RG_S. The algorithm terminated correctly, which means that state i has been reached ($i \in V_{SG}$) and $\forall t \in T_{NC} \exists (m, t, m') \in E_{SG}$. In the following we will prove that under these assumptions the properties 1–6 are satisfied by SG.

Property 1 holds because it has been checked in step iii) that node i is element of V_{SG}.
Property 2 holds because the subgraph has been cleaned in step iv) such that it only contains nodes which are reachable from i.
Property 3 holds because step ii.c) of the algorithm colored non-controllable state transitions either by bundle (all transitions of one non-controllable choice) or none of that choice.
Property 4 was tested in step v) and vi).
Property 5 trivially holds, because state o has been colored initially.
Property 6 holds because it has been checked in step v) that o is reachable from all states in V_{SG}.

We now prove the reverse direction: If the system $S = (PN, i)$ is non-controllable choice robust then the algorithm terminates correctly. We know there exist a winning strategy for the workflow controller thus a subgraph of the reachability graph for which the properties mentioned above hold. Let SG_W be the maximal winning strategy for the workflow controller.

The algorithm operates on the reachability graph RG_S and computes a subgraph SG_A. We will prove now that the computed subgraph SG_A coincides with SG_W: $SG_A = SG_W$. If this holds, the algorithm terminates correctly because then SG_A accomplishes the test criteria in step iii), v) and vii).

Proof. We will prove this direction indirectly. Assume the algorithm terminates but results in a subgraph SG_A which does either contains too few or to many elements compared to SG_W. The case where SG_W and SG_A are incomparable can not occur, because already at the initialization they share node o.

Assumption I: Algorithm terminates, but $SG_A \subset SG_W$ This means the algorithm stopped before coloring all appendant state transitions. It therefore exists at least one (finite) path $(m_0, t_1, m_1)....(m_{n-1}, t_n, m_n)$ in SG_W which does not exist in SG_A. The path leads into already colored area: $m_n \in V_{SG_A}$. The transition t_n labeling the last state transition of that path either belongs to T_{NC} or not. It is not the case that $t_n \notin T_{NC}$ because than step ii.b) of the algorithm could have been executed again. The transition t_n must therefore belong to $T_{NC} : t_n \in T_{NC}$. Lets consider an arbitrary transition t'_n belonging to the same non-controllable choice: $t_n, t'_n \in C_i$. While the choice satisfies the free choice property, it follows that t'_n is enabled in m. Taking property 3 into account it exists $m'_n \in V_{SG_W}$ and $(m_{n-1}, t'_n, m'_n) \in E_{SG_W}$. From property 6 follows that there exist a path from m'_n to o in SG_W, which means that m must be an element of $Pred_{RG}(o)$. This is the prerequisite to apply step ii.c) of the algorithm. In both cases the algorithm does not terminate which contradicts the assumption. The case that in step iv) elements are withdrawn from SG_A that do belong to SG_W is not possible, because property 2 states that for all elements $m \in V_{SG_W}$ exists a path from i to m.

Assumption II: Algorithm terminates, but $SG_W \subset SG_A$ This means during the first part of the algorithm (in step ii.b) and ii.c)) to many elements have been colored. Thus elements which later on did not get connected to i and hence do not exist in SG_W. To avoid this step iv) of the algorithm has been introduced. Here elements that do not belong to SG_W are cleaned up.

We therefore also proved the reverse direction that the algorithm terminates correctly if the system is non-controllable choice robust. Hence we conclude that Proposition 1 holds. □

5 Summary and Future Work

The contribution of this paper is the introduction of a new quality measure for workflow models, namely *non-controllable choice robustness*. The criterion describes whether a modeled process may react robust to environmental interactions. Beside *soundness* and *relaxed soundness robustness* provides the application developer with a further means to check the reasonability of their workflow models.

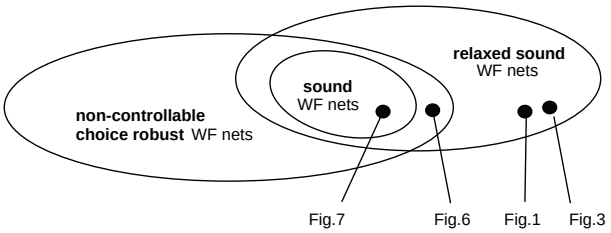


Fig. 11. Relation of the three different criteria

The three criteria are closely related. Soundness implies non-controllable choice robustness as well as relaxed soundness. The subset diagram in Figure 11 illustrates the relation of the three different criteria. Note that in the trivial case where a WF net does not contain any external choices relaxed soundness implies non-controllable choice robustness. Note that furthermore non-controllable choice robustness implies soundness if the WF net does only contain non-controllable choices. In the paper we find some examples that illustrate the different subsets depicted in Figure 11. The process in Figure 7 is sound and therefore also relaxed sound and robust. An examples for a process which is relaxed sound and robust but not sound is shown in Figure 6. Furthermore there exist processes that are relaxed sound but not robust and vice versa. Examples for the first case have been discussed in the beginning of this paper, cf. Figures 1 and 3. A process which is robust but not relaxed sound contains internal transitions that are not part of a sound firing sequence. This means the process can be controlled but some executions which are determined through internal choices are not chosen because they do not terminate properly.

At the end of the paper we presented an algorithm to check a bounded system for non-controllable choice robustness and proved its correctness and completeness. The implementation of the correctness check is an objective for future work. As for relaxed soundness we aim at enhancing the model checking tool LoLA [17] for this purpose.

Another objective of future work is to elaborate on automatic transformations between relaxed sound and robust systems and sound systems. We aim at providing rules for the generation of a sound process definition which could than be used as a model for the workflow controller and support the execution of only relaxed sound and robust processes at run time.

Acknowledgments

I thank Axel Martens for the inspiring discussions where some of the main ideas of this paper originated. I also thank Wil van der Aalst and Markus Schaal for helpful hints and constructive critic for improving the paper.

References

1. W. M. P. van der Aalst. Verification of Workflow Nets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *LNCS*, pages 407–426. Springer-Verlag, Berlin, 1997. 122, 123
2. W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. 122, 123, 124, 125, 127
3. W. M. P. van der Aalst and T. Basten. Inheritance of Workflows - An approach to tackling problems related to change. Computing Science Reports 99-06, University of Colerado Boulder, 1999. 121, 122
4. W. M. P. van der Aalst, B. Kiepuszewski, A. ter Hofstede, A. Barros, and B. Kiepuszewsk. Advanced Workflow Patterns. In O. Etzion and P. Scheuremann, editors, *Proceedings Seventh IFCIS International Conference on Cooperative Information System (CoopIS 2000)*, volume 1901 of *LNCS*, pages 18–29. Springer-Verlag, September 2000. 122, 127
5. N. R. Adam, V. Atluri, and W.-K. Huang. Modeling and Analysis of Workflows Using Petri Net. *Journal of Intelligent Information System, Special Issue on Workflow and Process Management*, 10(2):131–158, 1998. 123
6. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems*, pages 1–20, 1994. 130
7. J. Dehnert. Four Systematic Steps Towards Sound Business Process Models. In Weber et al. [19], pages 55–64. 125
8. J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K.L. Dittrich, A. Geppert, and M.C. Norrie, editors, *Advanced Information System Engineering, CAISE 2001*, volume 2068 of *LNCS*, pages 157–170. Springer Verlag, 2001. 125
9. W. Derks, J. Dehnert, P. Grefen, and W. Jonker. Customized atomicity specification for transactional workflow. In H. Lu and S. Spaccapietra, editors, *Proceedings of the Third International Symposium on Cooperative Database Systems and Applications (CODAS'01)*, pages 155–164. IEEE Computer Society, April 2001. 125
10. J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995. 127
11. R. Eshuis and R. Wieringa. A Comparison of Petri Net and Activity Diagram Variants. In Weber et al. [19], pages 93–104. 121, 122
12. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992. 131
13. A. Martens. Modeling Workflow in Virtual Enterprises. In Weber et al. [19], pages 157–162. 121, 122
14. R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993. 130, 135
15. A. Nerode, A. Yakhnis, and V. Yakhnis. Concurrent Programs as Strategies in Games. In *LFROMCS: Logic from Computer Science: Workshop-Proceedings*. Springer-Verlag, 1992. 135
16. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89. Proceedings of the sixteenth annual ACM symposium on Principles of programming languages*, pages 179–190, New York, 1989. ACM Press. 130
17. K. Schmidt. Lola: A low level analyser. In *Proc. Int. Conf. Application and Theory of Petri net*, volume 1825 of *LNCS*, pages 465–474, 1999.

18. W. Thomas. On the synthesis of strategies in infinite games. In *Symposium on Theoretical Aspects of Computer Science*, pages 1–13, 1995. 139 130, 131, 135
19. H. Weber, H. Ehrig, and W. Reisig, editors. *Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*. Fraunhofer Gesellschaft ISST, September 2001. 140

Real-Time Synchronised Petri Nets

Giovanna Di Marzo Serugendo[1], Dino Mandrioli[2],
Didier Buchs[3], and Nicolas Guelfi[4]

[1] Computer Science Department, University of Geneva
CH-1211 Geneva 4, Switzerland
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano
Milano 20133, Italy
[3] LGL-DI, Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland,
[4] Department of Applied Computer Science, IST - Luxembourg University of Applied Science
L-1359 Luxembourg-Kirchberg

Abstract. This paper presents the combination of two well established principles: the CO-OPN synchronisation mechanism, and the Merlin and Farber time Petri nets. Real-time synchronised Petri nets systems are then defined such that a Petri net is an object that can ask to be synchronised with another net, and whose transition firing is constrained by relative time intervals. Our proposal enables to define complex systems with compact specifications, whose semantics is given through a small set of Structured Operational Semantics (SOS) rules. The applicability of the new model is shown by applying it to a traditional benchmark adopted in the literature of real-time systems.

Keywords: CO-OPN, Petri nets, real-time, inhibitor arcs.

1 Introduction

This paper is a first step to enhance a high-level class of Petri nets with real-time constraints. Starting from a simplified version of the CO-OPN [4] language, where Petri nets are able to request synchronisation with each other, we have augmented the syntax and semantics with time intervals attached to the transitions in a way similar to that of Merlin and Farber nets [11].

A real-time synchronised Petri nets specification is *object-based*, i.e., it is made of a fixed number of objects that exist since the beginning of the system. An object is a real-time Petri net with inhibitor arcs. Such a net has two kinds of transitions: external transitions, called *methods*, and internal ones simply called *transitions*. Methods and transitions may request a *synchronisation* with methods, provided no cycles are formed. A method `m` that requests synchronisation with `m'` can fire only if `m'` can fire simultaneously. However, neither methods nor transitions can request a synchronisation with a transition.

Inhibitor arcs [10] provide a symmetric enabling of methods and transitions wrt usual pre-conditions: a transition cannot fire if a place connected to it

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 142–162, 2002.
© Springer-Verlag Berlin Heidelberg 2002

through an inhibitor arc contains tokens in a number exceeding the label of the arc. In addition, inhibitor arcs can be used as a priority mechanism among methods and transitions. Such a mechanism is quite useful to achieve time predictability in real-time systems. Time stamped *black tokens* are used for populating places.

A *time interval* is attached to any method and any transition. The firing of a method or a transition is considered to be instantaneous, it takes place within a time interval that is relative to the time when the method or transition becomes enabled (wrt the pre-set).

Figure 1 shows a simple real-time synchronised Petri net system made of two objects (two Petri nets) called **O1** and **O2**. Object **O1** contains a place `p1`, and a method `move`. The current marking of this place contains three black tokens stamped at time 0. Object **O2** contains a place `p2`, a method `put`, and a transition `processing`. Place `p2` is initially empty. There is an inhibitor arc, with label `0`, between place `p2` and method `put`. The dashed arrow from method `move` to method `put` states that method `move` requests a *synchronisation* with method `put` whenever it fires. In this particular case, the synchronisation corresponds to the classical fusion of transitions, even if its asymmetry makes it more general. Method `move` has the time interval `[5..15]` attached, method `put` has the time interval `[2..10]` attached, and transition `processing` the time interval `[1..9]`.

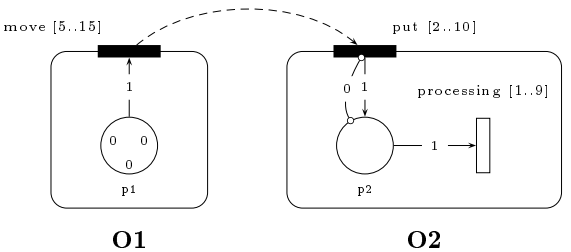


Fig. 1. A Real-time Synchronised Petri net

The intuitive semantics of this real-time synchronised Petri net system is the following: (1) method `move` requires to be synchronised with `put`, therefore `move` is enabled if its pre-condition is satisfied, and if `put` is enabled; (2) whenever `move` fires, `put` fires simultaneously, i.e., whenever `move` fires, a token is added in place `p2`. It is worth noting that `put` can fire alone without the firing of `move`; (3) the time interval attached to `move` means that `move` must fire instantaneously in the interval given by: `5` time slots after it becomes enabled, and `15` time slots after it becomes enabled. In addition, it *must* fire at the latest `15` time slots after it becomes enabled; (4) similarly, method `put` and transition `processing` must fire in the relative time interval `[2..10]`, and `[1..9]` respectively; (5) `move` requires to be synchronised with `put`, and their respective time intervals must be respected, this means that `move` and `put` must fire simultaneously in an interval corresponding to the intersection of the time interval of `move`, and that of `put`,

i.e., `[5..10]` (since both are enabled at time 0); (6) the inhibitor arc associated to `put` means that this method can fire only if the number of tokens in place `p2` is less or equal 0. Therefore, transition `processing` has a higher priority wrt method `put`. Indeed, method `put` cannot fire, if transition `processing` has not fired before, and thus emptied place `p2`. A second firing of method `put` can occur as soon as transition `processing` fires, but not before.

The main contribution of this paper consists in the definition of a compact syntax and semantics of these real-time synchronised Petri nets. The paper is structured in the following manner: section 2 defines the syntax of real-time synchronised Petri nets, section 3 describes their semantics, and section 4 gives an applicative example. A full version of this paper can be found in [5]. It includes a more detailed analysis of some intricacies in the semantics of the model.

2 Syntax

A *Petri net with inhibitor arcs* is a Petri net with two kinds of transitions: external ones, called methods, and internal ones, simply called transitions. Three kinds of arcs between places and methods/transitions are defined: input, output, and inhibitor arcs. Input, output arcs are traditional pre- post-conditions of nets. Inhibitor arcs prevent the firing of a single method or transition if the number of tokens in the place is greater than the number of tokens associated with the arc.

Definition 1. *Petri Net with Inhibitor Arcs.*
A Petri Net with inhibitor arcs is given by a 6-tuple $(P, M, T, Pre, Post, In)$ *where:* P *is a finite set of places;* M *is a finite set of methods;* T *is a finite set of (internal) transitions;* $Pre, Post : M \cup T \rightarrow (P \rightarrow \mathbb{N}\backslash\{0\})$ *are total functions, they define traditional Petri nets arcs removing or inserting black tokens respectively; to every method or transition is associated a partial function that maps places to a positive natural number.* $In : M \cup T \rightarrow (P \rightarrow \mathbb{N})$ *is a total function defining inhibitor arcs; to every method or transition is associated a partial function that maps places to a natural number.*

Remark 1. If $Pre(m)(p)$ is undefined there is no pre-set condition between place p and method m. It is similar for $Post(m)(p)$ and $In(m)(p)$. If $Pre(m)(p)$ is defined, then $Pre(m)(p) = 0$ is not allowed (idem for $Post(m)(p)$). If $In(m)(p)$ is defined, then $In(m)(p) = 0$ means there is an inhibitor arc with weight 0 between place p and method m.

A *real-time Petri net* is a Petri net with inhibitor arcs having a time interval associated to every method and transition.

Definition 2. *Real-Time Petri Net.*
A Real-Time Petri Net is given by a pair $(O, Time)$ *where* $O = (P, M, T, Pre, Post, In)$ *is a Petri net with inhibitor arcs, and* $Time$ *is a total function that*

associates a time interval to every method and transition of O. $Time : M \cup T \rightarrow \mathbb{R}^+ \times (\mathbb{R}^+ \cup \infty)$, *is such that the following condition must hold:*

$$Time(m) = (t_1, t_2) \;\; \Rightarrow \;\; ((t_2 \geq t_1) \;\vee\; (Time(m) = (t, \infty))) \; .$$

We denote t_1 *by* $Time_{Inf}(m)$, *and* t_2 *by* $Time_{Sup}(m)$.

A real-time synchronised Petri nets system is a set of real-time Petri nets with a synchronisation mapping among them. A method or transition may request to be synchronised with two or more methods simultaneously (//), in sequence (. .), or in alternative ($\oplus$).

Definition 3. *Real-Time Synchronised Petri nets System.*
A Real-Time Synchronised Petri nets System is given by $Sys = (O_1, \ldots, O_n, Sync)$*:*

- $O_i = ((P_i, M_i, T_i, Pre_i, Post_i, In_i), Time_i)$, $1 \leq i \leq n$, *a real-time Petri net;*
- *a total function* $Sync : \cup_{i \in \{1,\ldots,n\}}(M_i \cup T_i) \rightarrow Sync_{Expr}$ *that defines for each method and transition* $m \in \cup_{i \in \{1,\ldots,n\}}(M_i \cup T_i)$ *a synchronisation expression.*

The following conditions must hold:

- $\forall i, j \in \{1, \ldots, n\}$ *then* $P_i \cap M_j = P_i \cap T_j = M_i \cap T_j = \emptyset$;
- $\forall i, j \in \{1, \ldots, n\}, i \neq j$ *then* $P_i \cap P_j = \emptyset$, $M_i \cap M_j = \emptyset$, *and* $T_i \cap T_j = \emptyset$;
- *the set* $Sync_{Expr}$ *of synchronisation expressions is the least set such that:*

$$\begin{aligned} \epsilon &\in Sync_{Expr} \\ \forall M_i, i \in \{1, \ldots, n\}, M_i &\subset Sync_{Expr} \\ e_1, e_2 \in Sync_{Expr} \Rightarrow \; e_1 \;//\; e_2 &\in Sync_{Expr} \\ e_1, e_2 \in Sync_{Expr} \Rightarrow \; e_1 \;..\; e_2 &\in Sync_{Expr} \\ e_1, e_2 \in Sync_{Expr} \Rightarrow \; e_1 \oplus e_2 &\in Sync_{Expr} \; , \end{aligned}$$

  where ϵ *stands for the empty synchronisation. We write "*m with e*" to denote* $Sync(m) = e$*;*
- *the Sync function must ensure that a method does not synchronise with itself, and that the chain of synchronisations does not form cycles.*[1]

We denote by P the union of all sets of places P_i of a real-time synchronised Petri nets system. Similarly, we denote M, and T, the union of all methods, and all transitions respectively. We denote by Pre, $Post$, In, and $Time$ the extension of the pre-conditions, post-conditions, inhibitor arcs and time intervals to the Petri nets system.

Every token of the net is stamped with its arrival time. Several tokens may arrive in a place at the same time, as a result of the post-condition. The marking of a real-time synchronised Petri nets system is then a mapping that associates to every place a multiset of non-negative real numbers.

[1] For simplification purposes, we impose this limitation in order to prevent infinite behaviour.

Definition 4. *Marking, Set of Markings.*
Let $Sys = (O_1, \ldots, O_n, Sync)$ be a real-time synchronised Petri nets system. A marking is a total mapping:

$$mark : P \rightarrow [\mathbb{R}^+] \ .$$

We denote by $Mark$ the set of all markings of Sys.

A multiset of $\mathbb{R}^+$ is given by a function $f \in [\mathbb{R}^+]$ such that $f : \mathbb{R}^+ \rightarrow \mathbb{N}$ evaluates to zero, except on a finite number of cases (thus the number of tokens in a place is finite). Here, $f(t) = j$ means that j tokens arrived at time t. We denote by $\emptyset$ the empty multiset ($\emptyset(t) = 0$, $\forall t$), and $\{t_1, t_1, t_2, t_2, t_2\}$ a multiset containing two tokens arrived at time t_1, and three tokens arrived at time t_2. It is worth noting that $mark(p)(t) = j$ means that place p contains (among others) j tokens stamped with time t.

The sum of two markings returns, for every place, a new multiset made of the union of the two original multisets, where multiple occurrences of the same time stamp are taken into account.

Definition 5. *Sum of Markings.*
Let $Sys = (O_1, \ldots, O_n, Sync)$ be a real-time synchronised Petri nets system, and $Mark$ be the set of all markings of Sys. The sum of two markings is given by a mapping $+_{Mark} : Mark \times Mark \rightarrow Mark$ such that:

$$(mark_1 +_{Mark} mark_2)(p) = mark_1(p) +_{[\mathbb{R}^+]} mark_2(p) \ .$$

For every $t \in \mathbb{R}^+$, $(mark_1(p) +_{[\mathbb{R}^+]} mark_2(p))(t) = mark_1(p)(t) + mark_2(p)(t)$.
In the rest of this paper we simply note $+$ instead of $+_{Mark}$, and $+_{[\mathbb{R}^+]}$.

To every marking corresponds an *unstamped* marking, which returns for every place p the number of tokens present in the place regardless of their arrival time.

Definition 6. *Unstamped Markings.*
Let $Mark$ be the set of all markings. The Unstamped Markings are given by the the total mapping:

$$U : Mark \rightarrow (P \rightarrow \mathbb{N}) \ ,$$

where $U(mark)$ is a total mapping, s.t.

$$U(mark)(p) = \begin{cases} \sum_{t \in K_p} mark(p)(t) \\ 0, \textit{if } mark(p)(t) = 0, \forall t \in \mathbb{R}^+ \end{cases}$$

where $K_p = \{t \in \mathbb{R}^+ \mid mark(p)(t) > 0\}$.

$U(mark)$ returns for every place p the number of tokens present in the place. The sum is finite since the multiset $mark(p)$ has only a finite number of elements (K_p is the finite carrier set of $mark(p)$).

Definition 7. *Initial Marking.*
Let $Sys = (O_1, \ldots, O_n, Sync)$ *be a real-time synchronised Petri nets system. An initial marking is a marking,* $mark_k : P \rightarrow [\mathbb{R}^+]$*, such that for every* $p \in P$*:*

$$mark_k(p)(0) \geq 0$$
$$mark_k(p)(t) = 0, \forall t > 0 \, .$$

An initial marking is such that a place p contains tokens stamped at time 0. The places do not contain tokens stamped with a time greater than time 0.

Definition 8. *Marked Real-Time Synchronised Petri Nets System.*
A marked Synchronised Petri nets system is a pair $(Sys, mark_k)$ *where* Sys *is a real-time synchronised Petri nets system, and* $mark_k$ *is an initial marking for* Sys*.*

Figure 1 is the graphical notation of the marked real-time synchronised Petri nets system $(Sys, mark_k)$ given by:

$$\begin{aligned}
Sys &= (O_1, O_2, Sync) \\
O_1 &= (\{p1\}, \{\texttt{move}\}, \emptyset, Pre_1, Post_1, In_1, Time_1), \\
O_2 &= (\{p2\}, \{\texttt{put}\}, \{\texttt{processing}\}, Pre_2, Post_2, In_2, Time_2) \\
Pre_1(\texttt{move})(\texttt{p1}) &= 1, Pre_2(\texttt{processing})(\texttt{p2}) = 1 \\
Post_2(\texttt{put})(\texttt{p2}) &= 1, In_2(\texttt{put})(\texttt{p2}) = 0 \\
Time_1(\texttt{move}) &= (5, 15), Time_2(\texttt{put}) = (2, 10), Time_2(\texttt{processing}) = (1, 9) \\
Sync(\texttt{move}) &= \texttt{put}, Sync(\texttt{put}) = \epsilon, Sync(\texttt{processing}) = \epsilon \\
mark_k(\texttt{p1}) &= \{0, 0, 0\}, mark_k(\texttt{p2}) = \emptyset \, .
\end{aligned}$$

3 Semantics

Real-time synchronised Petri nets are a timed extension of a simplified version of CO-OPN/2 nets [2]. The establishment of their semantics follows that of CO-OPN/2: it is based on the use of *Structured Operational Semantics* (SOS) rules, similar to those of CO-OPN/2, but adapted to the real-time constraints.

We first build, using an initial set of rules, a *weak transition system* that contains transitions belonging to the weak time semantics (an enabled transition may not fire even if the time of occurrence elapses). Second, on the weak transition system, we apply a condition that enables to retain only those transitions that belong to the strong time semantics (an enabled transition *must* fire when the time of occurrence elapses) [6]. We obtain what we call the *strong transition system*. Third, on the strong transition system, we apply another set of rules (taking into account synchronisations) that enables us to obtain an *expanded transition system*. Then, we retain only those transitions necessary for the (observable) *strong time semantics*. Finally, in some cases, it is more valuable to consider a subset of the *strong time semantics* representing what we are actually interested to observe. We call this subset, the *system view semantics*.

Let us first give some preliminary definitions. An observable event is one of the following: the firing of a method, the firing of a transition, or the parallel (//) or sequence (..) firing of two observable events, or the alternative ($\oplus$) between two observable events.

Definition 9. *Observable Events.*
Let Sys be a real-time synchronised Petri nets system. The set of observable events of Sys, denoted by Obs_{Sys}*, is the least set such that:*[2]

$$M \cup T \subset Obs_{Sys}$$
$$e_1, e_2 \in Obs_{Sys} \Rightarrow e_1 \;//\; e_2 \in Obs_{Sys}$$
$$e_1, e_2 \in Obs_{Sys} \Rightarrow e_1 \;..\; e_2 \in Obs_{Sys}$$
$$e_1, e_2 \in Obs_{Sys} \Rightarrow e_1 \oplus e_2 \in Obs_{Sys} .$$

An event is any observable event, but also an event of the form "m with e", where the synchronisation is explicitly required.

Definition 10. *Events.*
Let Sys be a real-time synchronised Petri nets system. The set of events of Sys, denoted by Event, is the least set such that:

$$e \in Obs_{Sys} \Rightarrow e \in Event$$
$$e = m \,\text{with}\, e',\ m \in M \cup T, \text{and } e' \in Sync_{Expr} \Rightarrow e \in Event .$$

Transition systems for real-time synchronised Petri nets are made of 4-tuples, made of two markings, an event (not necessarily an observable one), and a time of occurrence.

Definition 11. *Transition System.*
Let Sys be a real-time synchronised Petri nets system. A transition system, trs, for Sys is such that:

$$trs \subseteq Mark \times Event \times Mark \times \mathbb{R}^+ .$$

We represent a 4-tuple $(mark_1, e, mark_2, t)$ *by* $mark_1 \xrightarrow[t]{e} mark_2$.

3.1 Weak Transition System

The rules for constructing the weak transition system of a real-time synchronised Petri net are given by rules BasicBeh and BasicSyncBeh formally described in Definition 12 below.

BasicBeh covers the case of the firing at time t of a single transition or method m that does not require any synchronisation. From a given marking

[2] an observable event has the same structure as a synchronisation expression. However a transition may appear in an observable event, while only methods are part of a synchronisation expression.

$mark_1$, the rule enables to compute the new marking after the firing of transition m alone. The 4-tuple is produced if several conditions are met: (1) the inhibitor arc condition is satisfied ($In(m) \geq U(mark_1)$); (2) the time of occurrence t is in the absolute time interval. The absolute time interval is computed from the relative one, given by $Time(m) = (Time_{Inf}(m), Time_{Sup}(m))$, and by the greatest time of arrival of the tokens that will be removed ($\max(mark)$);[3] (3) the number of tokens that will be removed must match the pre-condition, i.e. $U(mark) = Pre(m)$; (4) the pre-condition is satisfied ($mark_1 \geq mark$); (5) the new marking, after the firing of m is obtained by removing $mark$ from marking $mark_1$, and inserting new tokens stamped at time t ($Post(m)_t$).

BasicSyncBeh covers the case of the firing of a single transition or method that requires to be synchronised with some other methods, given by synchronisation expression e. The resulting 4-tuple is the same as rule BasicBeh, except the event part, which is of the form "m with e". The new marking is obtained from $mark_1$ by considering the firing of m alone (without e). Such tuples will be exploited in Section 3.3 to define the semantics of synchronisation. Indeed, the "with e" part of the event serves as a hook for combining transitions.

Definition 12. *Rules, Weak Transition System.*
Let Sys be a real-time synchronised Petri nets system. The weak transition system, denoted by trs_{weak}, is the set obtained by the application of the inference rules BasicBeh and BasicSyncBeh to $Mark$. In these rules: $m \in M \cup T$, $e \in Sync_{Expr}$, and $mark_1, mark \in Mark$ are markings.

$$\text{BasicBeh}\ \frac{\begin{array}{l} Sync(m) = \epsilon \\ In(m) \geq U(mark_1) \\ Time_{Inf}(m) + \max(mark) \leq t \leq Time_{Sup}(m) + \max(mark) \\ U(mark) = Pre(m) \\ mark_1 \geq mark \end{array}}{mark_1 \xrightarrow[t]{m} mark_1 - mark + Post(m)_t}$$

$$\text{BasicSyncBeh}\ \frac{\begin{array}{l} Sync(m) = e \\ In(m) \geq U(mark_1) \\ Time_{Inf}(m) + \max(mark) \leq t \leq Time_{Sup}(m) + \max(mark) \\ U(mark) = Pre(m) \\ mark_1 \geq mark \end{array}}{mark_1 \xrightarrow[t]{m \text{ with } e} mark_1 - mark + Post(m)_t}.$$

In the above rules, the following conventions are used:

- $In(m) \geq U(mark_1)$ holds if $In(m)(p) \geq U(mark_1)(p)$ for every p where $In(m)$ is defined;

[3] Marking $mark$ is a way of representing the pre-condition in a stamped form; it stands for the marking that will be removed from the places when the transition fires

- $U(mark) = Pre(m)$ holds if $U(mark)(p) = Pre(m)(p)$ for every p where $Pre(m)$ is defined, and $U(mark)(p) = 0$, otherwise;
- $\max(mark) = \max(\{0\}, \cup_{p \in P} K_p)$.
  It is the greatest time of arrival of the tokens that will be removed (i.e., tokens in $mark$). Indeed, $K_p = \{t \in \mathbb{R}^+ \mid mark(p)(t) > 0\}$ gives the time stamps of tokens in $mark$. Whenever $Time_{Sup}(m) = \infty$, then $Time_{Sup}(m) + \max(mark) = \infty$;
- $Post(m)_t : P \rightarrow [\mathbb{R}^+]$ is a marking such that all tokens are stamped at time t:

$$Post(m)_t(p)(t) = Post(m)(p)$$
$$Post(m)_t(p)(t') = 0, \forall t' \neq t \ ;$$

- $mark_1 \geq mark$ holds if $mark_1(p)(t) \geq mark(p)(t)$, for every $p \in P$, and $t \in \mathbb{R}^+$.

Rules of Definition 12 provide the weak time semantics since nothing forces enabled methods (or transitions) to fire within the given time interval.

Remark 2. The weak transition system is simply a set of 4-tuples. Trying, at this point, to make sequences of transitions based on markings may lead to paths where time goes backward. Therefore, we do not consider building paths at this stage.

3.2 Strong Transition System

The strong transition system is obtained as a subset of the weak transition system, by applying a condition that removes from the weak transition system, every transition, of the form $mark_1 \xrightarrow[t]{e} mark_2$, for which there exists another transition $mark_1 \xrightarrow[t']{e'} mark'_2$ that should have fired before $(t' < t)$.

Definition 13. *Strong Transition System.*
Let Sys be a real-time synchronised Petri nets system, and trs_{weak} be its weak transition system. The strong transition system, denoted by trs_{strong}, is the maximal subset of trs_{weak} such that:

$$\forall (mark_1 \xrightarrow[t]{e} mark_2) \in trs_{strong} \ \Rightarrow \ Cond(mark_1, mark_2, t) \text{ holds } .$$

Condition $Cond(mark_1, mark_2, t)$ is such that:

$$\begin{aligned} Cond(mark_1, mark_2, t) :=& \nexists\, (m', mark', t') \in (M \cup T) \times Mark \times \mathbb{R}^+ \text{ s.t.} \\ & t' < t \wedge \\ & t' = (Time_{Sup}(m') + \max(mark')) \wedge \\ & ((mark_1 \xrightarrow[t']{m'} mark_1 - mark' + Post(m')_{t'}) \in trs_{weak} \vee \\ & (mark_1 \xrightarrow[t']{m' \text{ with } e'} mark_1 - mark' + Post(m')_{t'}) \in trs_{weak}) \ . \end{aligned}$$

Remark 3. Condition *Cond* guarantees the strong time semantics: a transition m' *must* fire at time t' if it is enabled at t', and if time t' is the maximal bound of firing of the transition ($t' = Time_{Sup}(m') + \max(mark')$). However, when two transitions or more reach their maximal bound of firing at the same time, *Cond* does not apply, and one of the transitions may still disable the other.

3.3 Expanded Transition System

The expanded transition system is obtained from the strong transition system by adding tuples regarding synchronisation, simultaneity, alternative and sequence.

Sync handles the case of the synchronisation. From two transitions, one with a requested synchronisation, and one with the corresponding synchronisation that occur *at the same time* t, the rule produces a transition, occurring at t, where the synchronisation expression is abstracted. The new marking takes into account the effects of the simultaneous firing of m and e, but in the produced transition, the observable event m replaces "m with e". This rule produces transitions where only the firing of m is observable, but the result of the firing of m takes into account the behaviour of e. *Sim* handles the case of simultaneity of observable events. From two transitions: one for e_1 and one for e_2 that occur *at the same time* t, it builds the transition for event $e_1 \,//\, e_2$. *Alt.1* corresponds to the alternative case where e_1 fires. *Alt.2* corresponds to the case where e_2 fires. *Seq* defines transitions for sequential events: from two transitions whose final and initial marking correspond, and whose time of occurrence of the second is greater or equal to the time of occurrence of the first one, the rule produces the transition corresponding to their sequence. Definition 14 formally describe these rules.

Definition 14. *Rules, Expanded Transition System.*
Let Sys be a real-time synchronised Petri nets system, and trs_{strong} *its strong transition system. The expanded transition system, denoted by* trs_{expand}*, is the least set obtained by the successive application, to* trs_{strong}*, of the inference rules Sync, Sim, Alt.1, Alt.2, and Seq below. In these rules:* $m \in M \cup T$*,* $e \in Sync_{Expr}$*,* $e_1, e_2 \in Obs_{Sys}$*, and* $mark_1, mark_2, mark'_1, mark'_2, mark \in Mark$ *are markings.*

$$\text{Sync}\ \frac{\begin{array}{l} In(m) \geq U(mark_1) + U(mark'_1) + Post^*(e) \\ In^*(e) \geq U(mark_1) + U(mark'_1) + Post(m) \\ mark_1 \xrightarrow[t]{m \text{ with } e} mark_2 \quad mark'_1 \xrightarrow[t]{e} mark'_2 \end{array}}{mark_1 + mark'_1 \xrightarrow[t]{m} mark_2 + mark'_2}$$

$$\text{Sim}\ \frac{\begin{array}{l} In^*(e_1) \geq U(mark_1) + U(mark'_1) + Post^*(e_2) \\ In^*(e_2) \geq U(mark_1) + U(mark'_1) + Post^*(e_1) \\ mark_1 \xrightarrow[t]{e_1} mark_2 \quad mark'_1 \xrightarrow[t]{e_2} mark'_2 \end{array}}{mark_1 + mark'_1 \xrightarrow[t]{e_1 \,//\, e_2} mark_2 + mark'_2}$$

$$\text{Alt.1}\ \frac{mark_1 \xrightarrow[t]{e_1} mark_2}{mark_1 \xrightarrow[t]{e_1 \oplus e_2} mark_2} \qquad \text{Alt.2}\ \frac{mark'_1 \xrightarrow[t]{e_2} mark'_2}{mark'_1 \xrightarrow[t]{e_1 \oplus e_2} mark'_2}$$

$$\text{Seq}\ \frac{mark_1 \xrightarrow[t]{e_1} mark'_1 \quad mark'_1 \xrightarrow[t']{e_2} mark_2 \quad t' \geq t}{mark_1 \xrightarrow[t]{e_1 \,..\, e_2} mark_2}\ .$$

In the above rules, $In^*(e)$ stands for the minimal value associated to an inhibitor arc of a method or transition that takes part in the behaviour of e. $Post^*(e)$ stands for the sum of the post-conditions of all methods and transition taking part in e. Therefore, the conditions on the inhibitor arcs implies that the strongest condition applies. Formal definitions of $In^*(e)$ and $Post^*(e)$ are given in [5].

Remark 4. According to rules *Sync*, and *Sim*, an event of the form e // $(e_1 \,..\, e_2)$, or e with$(e_1 \,..\, e_2)$ occurs only if both e and $(e_1 \,..\, e_2)$ occur at the same time. Intuitively, such events should occur if e and e_1 occur simultaneously, and e_2 occurs later. Therefore, in rule *Seq*, the time of occurrence of a transition whose event is of the form $e_1 \,..\, e_2$ is the time of occurrence of e_1.

As a consequence of this choice, rule *Seq* builds 4-tuples where the resulting marking may contain tokens stamped at a time which is over the time of firing of the whole transition. These tokens result from the firing of e_2 which actually occurs later. Such tokens are actually not available; they will take part in transition firings (pre- post-conditions) only when the time will have advanced. However, they are taken into account for inhibitor arc evaluation even though they are not actually available. If such situations are not desired, the use of inhibitor arcs combined with the sequential operator should be avoided.

3.4 Strong Time Semantics

Similarly to the weak transition system, the expanded transition system contains only 4-tuples, i.e., no paths are considered. The strong time semantics builds meaningful paths from the 4-tuples available in the expanded transition system. Therefore, the semantics of a real-time synchronised Petri nets system is obtained by retaining, from trs_{expand}, all the sequences of transitions containing: (1) observable events only (no m with e); (2) markings reachable from the initial marking on a path where time is monotonic. A path p is a sequence of 4-tuples. We denote $tail(p)$ the last 4-tuple of the path.

Definition 15. *Strong Time Semantics.*
Let $(Sys, mark_k)$ be a marked real-time synchronised Petri nets system, trs_{expand} be the expanded transition system obtained with the rules of Definition 14. The

strong time semantics of $(Sys, mark_k)$*, denoted by* Sem*, is the least set of paths such that:*

$$(mark_k \xrightarrow[t]{e} mark') \in trs_{expand} \ \wedge \ e \in Obs_{Sys} \Rightarrow (mark_k \xrightarrow[t]{e} mark') \in Sem$$

$$(mark'_1 \xrightarrow[t]{e} mark_2) \in trs_{expand} \wedge e \in Obs_{Sys} \wedge \exists\, p \in Sem \ s.t. \ tail(p) = (mark_1 \xrightarrow[t']{e'} mark'_1) \ \wedge \ t' \leq t \Rightarrow p\,(mark'_1 \xrightarrow[t]{e} mark_2) \in Sem\ .$$

Remark 5. If we want to obtain the weak time semantics, instead of the strong time semantics, we proceed from the weak transition system given by Definition 12, then we apply the rules of Definition 14, without applying the condition *Cond*, and finally we apply Definition 15.

3.5 System View Semantics

In some cases, the whole observable strong time semantics is too vast, and we want to keep only a subset of behaviours in order to analyse them. We model a lot of behaviours, but we want to observe actually only few of them.

For this reason, in addition to the strong time semantics, we define the *System View Semantics* representing only those paths that we want to observe. We will see in the example how it is useful to not see part of the behaviour of some component.

The system view semantics is obtained from the strong time semantics by retaining from the transition system only the paths labelled with methods and transitions that we want to observe. Therefore, we need first to choose a set $View \subseteq (M \cup T)$ of methods and transitions that we want to observe. On the basis of $View$, we define the set of *observed* events, in a similar manner to the observable events.

Definition 16. *Observed Events.*
Let Sys *be a real-time synchronised Petri nets system. Let* $View \subseteq (M \cup T)$ *be the set of methods and transitions that we actually want to observe. The set of observed events of* Sys*, denoted by* $View_{Sys}$*, is the least set such that:*

$$View \subset View_{Sys}$$
$$e_1, e_2 \in View_{Sys} \ \Rightarrow \ e_1 \ // \ e_2 \in View_{Sys}$$
$$e_1, e_2 \in View_{Sys} \ \Rightarrow \ e_1 \ .. \ e_2 \in View_{Sys}$$
$$e_1, e_2 \in View_{Sys} \ \Rightarrow \ e_1 \oplus e_2 \in View_{Sys}\ .$$

An observed event has the same structure as an observable event. However only methods and transitions being part of the $View$ set can appear in an observed event.

The system view semantics is then obtained from the expanded transition system, trs_{expand}, by retaining the transitions containing observed events only, and whose markings are reachable from the initial marking.

Definition 17. *System View Semantics.*
Let $(Sys, mark_k)$ be a marked real-time synchronised Petri nets system, let $View \subseteq (M \cup T)$, trs_{expand} be the expanded transition system obtained with the rules of Definition 14. The system view semantics of $(Sys, mark_k)$, denoted by Sem_{View}, is given by the least set of paths such:

$$(mark_k \xrightarrow[t]{e} mark') \in trs_{expand} \ \wedge \ e \in View_{Sys} \Rightarrow (mark_k \xrightarrow[t]{e} mark') \in Sem_{View}$$

$$(mark'_1 \xrightarrow[t]{e} mark_2) \in trs_{expand} \wedge e \in View_{Sys} \wedge \exists\, p \in Sem_{View} \ s.t.\ tail(p) = (mark_1 \xrightarrow[t']{e'} mark'_1) \ \wedge \ t' \leq t \Rightarrow p\,(mark'_1 \xrightarrow[t]{e} mark_2) \in Sem_{View}\ .$$

The system view semantics Sem_{View} is actually a subset of Sem, where we remove all the branches of the semantics tree labelled with events made with methods or transitions that are not part of $View$. Paths of Sem_{View} are made of transitions whose events are exclusively those that we want to observe.

It is important to note that even though a method `m` is not part of $View$, its behaviour is taken into account if it is requested for synchronisation by another method `m' ` $\in View$.

3.6 Example

Figure 2 shows a partial view of the tree of reachable markings of the strong time semantics of the real-time synchronised Petri nets system of Figure 1.

The initial marking is made of three tokens present at time 0 in place `p1`, and the empty place `p2`; it is denoted $\{0,0,0\}, \emptyset$. Method `move` has to occur between time 5 and time 10 (because of `put`), and method `put` has to occur between time 2 and time 10. In this example, the relative time intervals are also the absolute ones, since `move` and `put` are enabled at 0, and no token may arrive in their pre-set. The figure shows several cases. For instance, the firing of `move` occurring at time 5 leads to a new marking where a token has been removed from `p1`, and a token stamped at 5 is in `p2`. This corresponds to transition: $\{0,0,0\}, \emptyset \xrightarrow[5]{\text{move}} \{0,0\}, \{5\}$.

The formula below shows how the inference rules are applied in order to obtain this transition:

$$\dfrac{\dfrac{}{\{0,0,0\},\emptyset \xrightarrow[5]{\text{move with put}} \{0,0\},\emptyset}\,\text{BasicSyncBeh} \qquad \dfrac{}{\emptyset,\emptyset \xrightarrow[5]{\text{put}} \emptyset,\{5\}}\,\text{BasicBeh}}{\{0,0,0\},\emptyset \xrightarrow[5]{\text{move}} \{0,0\},\{5\}}\,\text{Sync}\,.$$

First, BasicBeh and BasicSyncBeh are applied. The obtained transitions are part of the weak transition system. Then rule Sync is applied on the two transitions. The resulting transition is then part of the strong time semantics, since it starts from the initial marking.

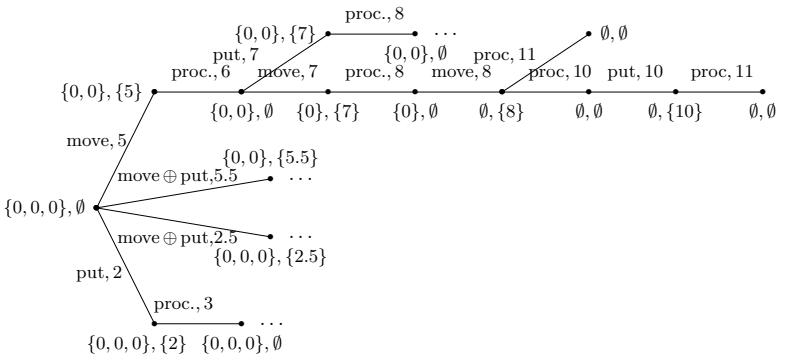


Fig. 2. Tree of reachable markings – Strong Time Semantics

If we come back to Figure 2, the firing of `move` must be followed by the firing of the `processing` transition, because of the inhibitor arc. Since `processing` is enabled at 5, it can fire in the time interval `[6..14]`. For instance, it fires at time 6, and `p2` becomes empty. At this point, `put` can fire alone, or `move` can fire (for instance at time 7), but still within their absolute time interval. After the second firing of `move`, there is necessarily a firing of `processing`, which can occur at time 8 or after. Then, we can have either a firing of `move` or a firing of `put`. For instance, `move` fires a last time at 8 (all tokens in the pre-set are consumed), then it is followed by `processing`. If `processing` occurs between 9 and 10, then `put` has to occur at the latest at 10, then `processing` occurs a last time at 11 or after. If `processing` occurs after 10, then `put` cannot fire since the current time is over its time interval. In both cases, the system has reached its end, no method or transition may fire. From the initial marking, it is also possible to fire `put` at time 2 for instance, thus producing a new marking $\{0,0,0\},\{2\}$; or to fire the alternative event `move` $\oplus$ `put`. The tree shows the case of `move` $\oplus$ `put` firing at 2.5, corresponding to the firing of `put`, or at 5.5, corresponding to the firing of `move`. It is not possible to fire event `move` // `put`. Since `move` requires the firing of `put`, an event such as `move` // `put` requires that `put` fires twice simultaneously. Method `put` *cannot* fire two or more times simultaneously because the inhibitor

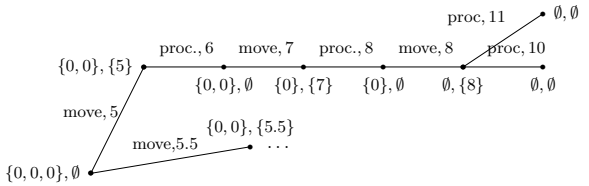


Fig. 3. Tree of reachable markings - System View Semantics

arc is set to 0. The tree depicted by Figure 2 is not complete: it is also possible to fire `move` at any time in the interval `[5..10]`; and to fire `put` at any time in the interval `[2..10]`.

Figure 3 shows the system view semantics, with $View = \{$`move`,`processing`$\}$. It is a subset of the tree of Figure 2, where method `put` alone cannot fire.

Remark 6. Zeno Behaviour. The semantics given above allows cases, where the number of sequential firings of a transition or a method, at a given time t, may be uncountable. These cases occur for transitions or methods having no pre-set, and no inhibitor arc, or when intervals [0..0] are used. It is recommended, to insert extra places with pre-sets, or inhibitor arcs, in order to prevent time stuttering.

4 An Applicative Example: The Railroad Crossing System

In this section we illustrate the applicability of the real-time synchronised Petri nets to practical cases through a fairly classical "benchmark" for real-time system formalisms, i.e., the Generalised Railroad Crossing (GRC) system [8].

The GRC system consists of one or more train trails which are traversed by a road. To avoid collisions between trains and cars a bar is automatically operated at the crossing. Let us call I the portion of train trails which crosses the road. To properly control the bar, sensors are placed on the trails to detect the arrival and the exit of trains: the arrival of a train must be signalled somewhat in advance wrt the train entering region I, whereas the exit is signalled exactly when a train exits I. We call R the portion of train trails included between the place where entering sensors are placed and the beginning of I (see Figure 4). All trains have a minimum and a maximum speed so that they take a minimum and a maximum, yet finite and non-null, time to traverse R and I.

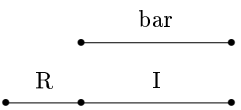


Fig. 4. Critical Section

The control of the bar operates as follows. Whenever a train enters R, this is detected and signalled by a sensor; similarly when a train exits I. If a train enters R and the bar is open (up), then a command is issued to the bar to close. This takes a fixed amount of time (γ). As soon as no more trains are in R or in I (this must be computed by the control apparatus on the basis of entering and exiting signals) the opening command is issued to the bar, which again takes γ time units to open (notice that, in this description, we assume that if a train enters R while the bar is opening, the control must wait until the bar is open before restarting closing it). The designer's job is to set system parameters (e.g., the length of R and the duration γ) in such a way that the following properties hold:

- Safety property: when a train is in I the bar is closed;
- Utility property: the bar is closed only for the time that is strictly necessary to guarantee the safety property.

Let us now formalise the GRC system through real-time synchronised Petri nets. Figure 5 shows two objects: `Train` and `Level Crossing`. The `Train` object represents the entry and the exit of trains into a critical region: a train enters into the critical region with method `entry`, it stays first in the section corresponding to R (place `p2`), for a certain amount of time (at least t_1). Then, it enters region I, represented by place `p3`, and finally it leaves the critical region with method `exit`. Several trains may be simultaneously in the critical region, however their entry is not simultaneous. Indeed method `entry` can fire more than once, since place `p1` contains always one token. However, method `entry` cannot fire twice or more simultaneously, and there is a delay of at least t'_1 between two trains entering the critical region. The fact that two trains may be simultaneously in region R or in region I depends also on the values of $t_1, t_2, t'_1, t''_1, t''_2$. Indeed, if $t_2 + t''_2 < t'_1$ then there will be at most one train in the critical region.

The `Level Crossing` object represents the behaviour of the critical region, i.e., the bar's behaviour: it must be up iff no train is currently in the critical region, or entering it.

Each time a train enters the critical region, the `signal_entry` method fires. This is due to the fact that `signal_entry` is requested by `entry`. The `signal_entry` method increases the number of tokens in place `Counter`, whose role is to count the number of trains that are currently in the critical region. If the barrier is up and if a train arrives in the critical region, transition `go_down` fires and the barrier begins to go down (place `mv_down`). After a certain amount of time γ, represented by transition `end_down`, the barrier is finally down (place `down`).

Each time a train leaves the critical region, by activating method `exit`, the `signal_exit` method fires simultaneously. This method simply decreases by one the number of trains that are currently in the critical region. As soon as there are no more trains in the critical region, i.e. place `Counter` is empty, transition `go_up` fires. Indeed, the inhibitor arc of weight 0 prevents the firing of `go_up` before all trains leave the critical region, and time interval [0..0] attached to `go_up` enforces `go_up` to fire as soon as it is enabled. The place `en` maintains the

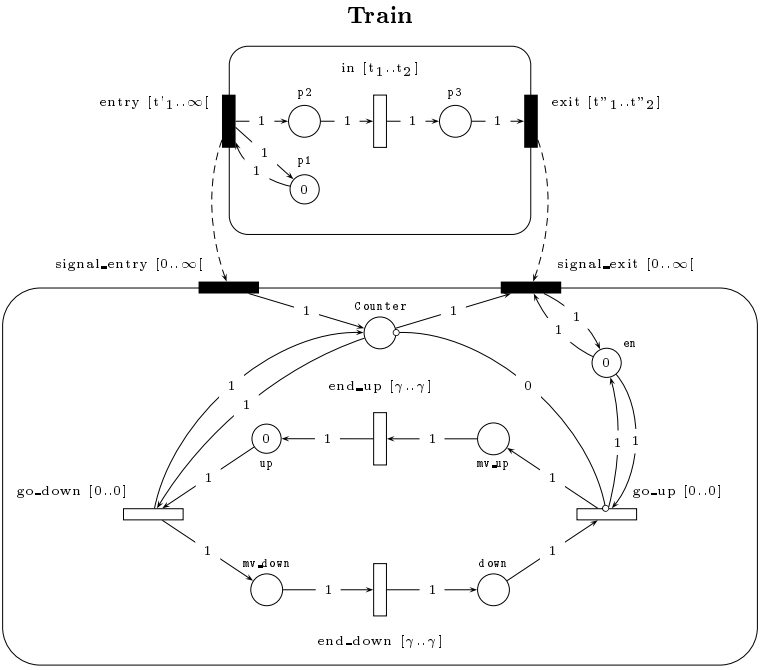


Fig. 5. Train and Level Crossing

time of exit of the last train. Then, the barrier begins to go up (place `mv_up`), and after a certain amount of time γ, represented by transition `end_up`, it arrives in the up position (place `up`).

When trains are in the critical region, and the barrier is already down, neither method `go_down` nor method `go_up` can fire. Therefore, the barrier remains down until all trains have leaved the critical region.

For the System view semantics we chose $View$ = {`entry`,`exit`,`go_down`, `end_down`,`go_up`,`end_up`}. We do not consider paths where methods `signal_entry` and `signal_exit` fire without being requested by methods `entry` and `exit` respectively. Indeed, train and level crossing are obviously two different objects. However, the level crossing is such that the commands activating the bar are issued from the trains, i.e., methods `signal_entry`, and `signal_exit` fire whenever a train enters R or exits I respectively, but these two methods should not fire alone. Let us now see how the safety property is satisfied.

The sequence of transitions (1) shows an incorrect sequence[4] of transitions wrt the safety property.

[4] for space purposes, markings are not shown, only transitions and time of firing are represented.

$$\{.\} \xrightarrow[t]{\texttt{entry}} \{.\} \xrightarrow[t]{\texttt{go_down}} \{.\} \xrightarrow[t+t_1]{\texttt{in}} \{.\} \xrightarrow[t+\gamma]{\texttt{end_down}} \{.\} \tag{1}$$

This sequence of transitions corresponds to the case a train entering region R at t (`entry`), and region I at $t+t_1$ (`in`) *before* the bar is down at $t+\gamma$ (`end_down`). This case occurs whenever $\gamma \geq t_1$. The first condition to impose on γ is then $\gamma < t_1$. The sequence of transitions (2) shows the correct sequence of the bar going down when one train enters R and $\gamma < t_1$. Sequence (2) shows as well the exit of the train at $t+t_1+t_1^{''}$ (`exit`), followed by the bar immediately beginning to go up, and finally reaching the up position $t+t_1+t_1^{''}+\gamma$ (`end_up`). Whenever a second train enters R at $t+t'$, after the bar is up, then the bar begins to go down normally, and the safety property is still satisfied.

$$\begin{aligned} &\{.\} \xrightarrow[t]{\texttt{entry}} \{.\} \xrightarrow[t]{\texttt{go_down}} \{.\} \xrightarrow[t+\gamma]{\texttt{end_down}} \{.\} \xrightarrow[t+t_1]{\texttt{in}} \{.\} \xrightarrow[t+t_1+t_1^{''}]{\texttt{exit}} \\ &\{.\} \xrightarrow[t+t_1+t_1^{''}]{\texttt{go_up}} \{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\texttt{end_up}} \{.\} \xrightarrow[t+t']{\texttt{entry}} \{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\texttt{go_down}} \{.\} \end{aligned} \tag{2}$$

It is similar, if a second train enters region R while the first train has not yet leaved I: the bar remains down until the second train leaves I. However, condition $\gamma < t_1$ is no longer sufficient, if the second train enters region R just after the exit of the first train, but before the bar is up. Indeed, the bar has received the signal to go up, and it will have to completely go up before being able to come down. The first train enters R at t, leaves I at $t+t_1+t_1^{''}$. The second train enters R at $t+t'$ ($t_1+t_1^{''} \leq t'$). Since the first train exits at $t+t_1+t_1^{''}$, and the second train is not yet in R, the bar begins to go up at $t+t_1+t_1^{''}$, reaches the up position at $t+t_1+t_1^{''}+\gamma$. Since in the meanwhile, the second train has entered R, the bar immediately begins to go down, and reaches the down position at $t+t_1+t_1^{''}+2\gamma$.

The incorrect case (3) occurs whenever the second train arrives after the exit of the first train: $t_1+t_1^{''} \leq t'$; and the bar is down (`end_down`) *after* the train has entered I (`in`): $t'+t_1 \leq t_1+t_1^{''}+2\gamma$, i.e., $t' \leq t_1^{''}+2\gamma$.

By combining these two conditions, we obtain the following equation: $t_1+t_1^{''} \leq t' \leq t_1^{''}+2\gamma$. It reduces to: $t_1+t_1^{''} \leq t_1^{''}+2\gamma$, and finally $t_1/2 \leq \gamma$. Therefore, in order to prevent the incorrect behaviour, γ must be such that: $\gamma < t_1/2$.

$$\begin{aligned} &\{.\} \xrightarrow[t]{\texttt{entry}} \{.\} \xrightarrow[t]{\texttt{go_down}} \{.\} \xrightarrow[t+\gamma]{\texttt{end_down}} \{.\} \xrightarrow[t+t_1]{\texttt{in}} \\ &\{.\} \xrightarrow[t+t_1+t_1^{''}]{\texttt{exit}} \{.\} \xrightarrow[t+t_1+t_1^{''}]{\texttt{go_up}} \{.\} \xrightarrow[t+t']{\texttt{entry}} \{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\texttt{end_up}} \\ &\{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\texttt{go_down}} \{.\} \xrightarrow[t+t'+t_1]{\texttt{in}} \{.\} \xrightarrow[t+t_1+t_1^{''}+2\gamma]{\texttt{end_down}} \{.\} \end{aligned} \tag{3}$$

In the correct case (4), the bar reaches the down position *before* the second train arrives in I.

$$\begin{aligned}
&\{.\} \xrightarrow[t]{\text{entry}} \{.\} \xrightarrow[t]{\text{go_down}} \{.\} \xrightarrow[t+\gamma]{\text{end_down}} \{.\} \xrightarrow[t+t_1]{\text{in}} \\
&\{.\} \xrightarrow[t+t_1+t_1^{''}]{\text{exit}} \{.\} \xrightarrow[t+t_1+t_1^{''}]{\text{go_up}} \{.\} \xrightarrow[t+t']{\text{entry}} \{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\text{end_up}} \\
&\{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\text{go_down}} \{.\} \xrightarrow[t+t_1+t_1^{''}+2\gamma]{\text{end_down}} \{.\} \xrightarrow[t+t'+t_1]{\text{in}} \{.\}
\end{aligned} \tag{4}$$

The utility property is simpler to see. Whenever the last train leaves I while the bar is already down, the inhibitor arc and the interval [0..0] attached to `go_up` guarantee that the bar begins to go up immediately when no more trains are in the critical region. Problems arise when the last train leaves I while the bar is *not yet* in the down position, i.e., transition `end_down` fires *after* the `exit` of the last train. The sequences of transitions (5) and (6) below show two occurrences of this case. Sequence (5) shows the case of a single train, entering R at t, and leaving I at $t+t_1+t_1^{''}$ before the bar reaches the down position. This is a special case of (1), it shows a worst behaviour, since the train not only enters I, but even leaves I before the bar is in the down position. Requiring $\gamma > t_1$ is sufficient to avoid this case.

$$\{.\} \xrightarrow[t]{\text{entry}} \{.\} \xrightarrow[t]{\text{go_up}} \{.\} \xrightarrow[t+t_1]{\text{in}} \{.\} \xrightarrow[t+t_1+t_1^{''}]{\text{exit}} \{.\} \xrightarrow[t+\gamma]{\text{end_down}} \{.\} \tag{5}$$

Sequence (6) shows a particular case of (3), when two trains are involved. As in (3) the second train enters R soon after the exit of the first train: $t_1 + t_1^{''} \leq t'$. Due to the bar going up and down, the second train leaves I before the bar is down: $t' + t_1 + t_1^{''} \leq t_1 + t_1^{''} + 2\gamma$, i.e., $t' \leq 2\gamma$.

$$\begin{aligned}
&\ldots\{.\} \xrightarrow[t+t_1+t_1^{''}]{\text{exit}} \{.\} \xrightarrow[t+t_1+t_1^{''}]{\text{go_up}} \{.\} \xrightarrow[t+t']{\text{entry}} \{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\text{end_up}} \\
&\{.\} \xrightarrow[t+t'+t_1]{\text{in}} \{.\} \xrightarrow[t+t_1+t_1^{''}+\gamma]{\text{go_down}} \{.\} \xrightarrow[t+t'+t_1+t_1^{''}]{\text{exit}} \{.\} \xrightarrow[t+t_1+t_1^{''}+2\gamma]{\text{end_down}} \{.\}
\end{aligned} \tag{6}$$

By combining both equations we have: $t_1 + t_1^{''} \leq t' \leq 2\gamma$, which reduces to: $\gamma \leq (t_1 + t_1^{''})/2$. Therefore, in order to avoid this we must enforce: $\gamma < (t_1 + t_1^{''})/2$. The requirement needed for the safety property: $\gamma < t_1/2$ is sufficient to ensure the utility property too.

The GRC example, though still rather simple, illustrates the suitability of the model for the description of real-time systems. First, modularisation is naturally achieved through the definition of several objects and the use of the synchronisation mechanism to formalise their interaction. Second, the use of inhibitor arcs allows to achieve a good level of generality without resorting to more sophisticated models. In our case inhibitors arcs allow a natural formalisation of the counter mechanism which is essential for a proper description of the system.

Notice that pure Petri nets allow only the modelling of simplified versions of the GRC system (e.g. the case where only one train can traverse the regions R and I per time) whereas in order to deal with the general case, more cumbersome formalisations are usually needed [8].

5 Related Works

The model of Communicating Time Petri Nets [3] is close to the one presented in this paper. It combines inhibitor arcs, attaches firing time intervals to transitions, and allows decomposition into modules. It differs in the composition of modules, which is realised through a message passing-based communication among modules. This work has also resulted in an analysis technique of the overall system based on an analysis of each individual modules. The combination of time and Petri nets modules has been also addressed in [9] which introduces the Time Petri Box calculus. Modules are Petri Boxes equipped with a dynamic firing time interval over a discrete time domain, and composition is realised by the means of several operators. The analysis method for single Time Petri nets of [1] is based on a "state-class" technique and allows to build a finite representation of the behaviour of the nets, enabling a reachability analysis similar to that of Petri nets.

6 Conclusion

We introduced *real-time synchronised Petri nets*, a class of high-level Petri nets that combines modularity and abstraction mechanisms (transactional view of synchronisations) proposed by CO-OPN, inhibitor arcs, and a delay time model for Petri nets (using relative time intervals). This paper does not address problems related to reachability analysis, even though the defined semantics constitutes a basis for elaborations of analysis techniques and tools. Efforts towards this direction have already been realised and an operational semantics is now available [12]. It relies on a symbolic technique that enables to build a finite representation of temporal constraints, which actually defines an uncountable number of state spaces and firings. Future work will concentrate on giving an axiomatisation to these nets in order to enable formal verification of logical properties (safety, liveness), and on pursuing further generalisation. Once the axiomatisation will be available, supporting theorem proving tools will be applicable to mechanise and make system analysis more robust [7].

References

1. B. Berthomieu. La méthode des Classes d'États pour l'Analyse des Réseaux Temporels - Mise en Oeuvre, Extension à la multi-sensibilisation. In *Modélisation des Systèmes Réactifs, MSR'2001.* Hermes, 2001. 161

2. O. Biberstein, D. Buchs, and N. Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha, F. De Cindio, and G. Rozenberg, editors, *Advances in Petri Nets on Object-Orientation*, volume 2001 of *LNCS*, pages 70–127. Springer-Verlag, 2001. 147
3. G. Bucci and E. Vicario. Compositional Validation of Time-Critical Systems Using Communicating Time Petri Nets. *IEEE Transactions on Software Engineering*, 21(12):969–992, 1995. 161
4. D. Buchs and N. Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering, Special Section on Formal Methods for Object Systems*, 26(7):635–652, July 2000. 142
5. G. Di Marzo Serugendo, D. Mandrioli, D. Buchs, and N. Guelfi. Real-time synchronised Petri nets. Technical Report 2000/341, Swiss Federal Institute of Technology (EPFL), Software Engineering Laboratory, Lausanne, Switzerland, 2000. 144, 152
6. M. Felder, D. Mandrioli, and A. Morzenti. Proving properties of real-time systems through logical specifications and Petri net models. *IEEE Transactions on Software Engineering*, 20(2):127–141, February 1994. 147
7. A. Gargantini and A. Morzenti. Automated Deductive Requirements Analysis of Critical Systems. *ACM TOSEM - Transactions On Software Engineering and Methodologies*, 10(3):255–307, July 2001. 161
8. C. Heitmeyer and D. Mandrioli, editors. *Formal methods for real-time computing.* John Wiley & Sons, 1996. 156, 161
9. M. Koutny. A Compositional Model of Time Petri Nets. In M. Nielsen and D. Simpson, editors, *International Conference on Application and Theory of Petri Nets 2000*, volume 1825 of *LNCS*, pages 303–322. Springer-Verlag, 2000. 161
10. C. A. Lakos and S. Christensen. A General Systematic Approach to Arc Extensions for Coloured Petri Nets. In R. Vallette, editor, *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *LNCS*, pages 338–357. Springer-Verlag, 1994. 142
11. P.M. Merlin and D.J. Farber. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, 1976. 142
12. S. Souksavanh. Operational Semantics for Real-Time Synchronized Petri Nets. Master's thesis, Swiss Federal Institute of Technology (EPFL), Software Engineering Laboratory, Lausanne, Switzerland, 2002. 161

Computing a Finite Prefix of a Time Petri Net*

Hans Fleischhack and Christian Stehno

Fachbereich Informatik, Carl von Ossietzky Universität Oldenburg
D-26111 Oldenburg
{fleischhack,stehno}@informatik.uni-oldenburg.de

Abstract. Recently, model checking of Petri nets based on partial order semantics w.r.t. temporal logic formulae has been extended to time Petri nets. In this paper, we present an improved algorithm for computing the McMillan-unfolding of a time Petri net which gives a finite representation of the partial order semantics and some experimental results of its implementation within the PEP tool.

1 Introduction

Model checking is a widely accepted method for proving properties of distributed systems but faces the problem of 'state explosion'. To tackle this problem, besides partial order reductions [12] or BDD-based techniques [7,19] also methods based on partial order semantics have been applied. The latter have proven especially successful when applied to systems with a high degree of asynchronous parallelism [5,9].

Safety critical applications often require verification of real time constraints, in addition to functional or qualitative temporal properties. For this task, model checking algorithms have been developed based on interleaving semantics (cf. e.g. [2,22]), but much less work has been done starting from partial order semantics. By extending McMillan's [20] technique of unfolding safe Petri nets to the class of safe time Petri nets a considerable step in this direction has been taken in [6].

In this paper, we present a simplified definition of the McMillan-unfolding of a time Petri net from which we derive an efficient algorithm for its actual computation, thereby also removing the restrictions of diverging time and of finite upper time bounds for transitions (cf. [6]) .

The key idea of the approach presented here consists in transforming time restrictions into net structure, i.e. representing them by additional places, transitions, and arcs. Following this approach, for a given time Petri net TN the time-expansion $EX(TN)$ is constructed as an ordinary P/T-net. Moreover, it is shown that a formula ϕ of a suitable temporal logic for time Petri nets is satisfied by TN iff it is satisfied by $EX(TN)$. Subsequently, the McMillan-unfolding $McM(TN)$ of TN is defined as (an abstraction of) $McM(EX(TN))$. As it turns

* This work has been partially supported by the the Procope projects BAT (Box Algebra with Time) and PORTA (Partial Order Real Time Semantics).

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 163–181, 2002.
© Springer-Verlag Berlin Heidelberg 2002

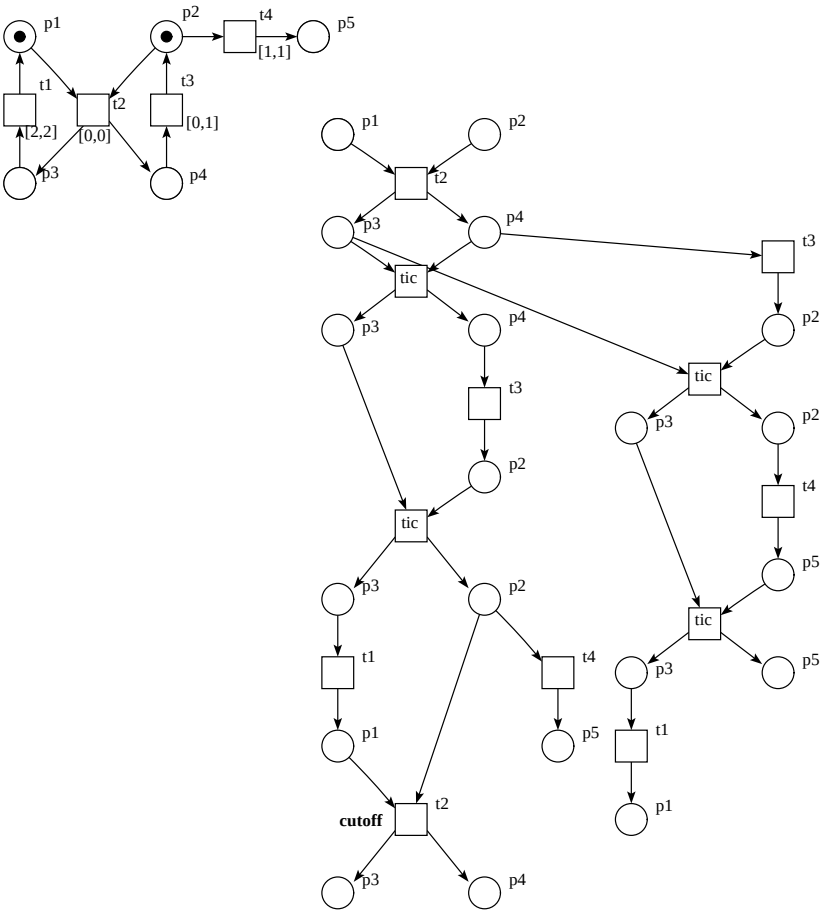


Fig. 1. A time Petri net and its timed behaviour

out, for the actual computation of $McM(TN)$ the explicit construction of the (complex) net $EX(TN)$ is not necessary.

Following this approach corresponding tools for P/T-nets may be reused for the causal analysis of time Petri nets. This applies especially to the model checking component of the PEP tool [5,23].

The paper is structured as follows: Section 2 provides the basic notions used throughout the paper. In Sect. 3 a new notion of bisimulation for time Petri nets and P/T-nets is introduced. Section 4 contains the definition of time expansion of time Petri nets. The new method for the actual computation of the McMillan-unfolding of a safe time Petri net is given in Sect. 5. Based on the implementation of this method we give some experimental results in Sect. 6. We conclude with discussing some related work (Sect. 7) and with remarks on further research (Sect. 8).

2 Basic Notions

In this section, partial order semantics and temporal logics for safe Petri nets is defined. A notion of time is added to P/T-nets and the temporal logics is extended to time Petri nets.

2.1 Petri Nets

A *net* is a triple $N = (P, T, F)$ consisting of a set P of *places*, a set T of *transitions*, a set $F \subseteq ((P \times T) \cup (T \times P))$, called *flow relation*, and such that $P \cap T = \emptyset$, ${}^\bullet p \neq \emptyset$ or $p^\bullet \neq \emptyset$ for all $p \in P$, and ${}^\bullet t \neq \emptyset \neq t^\bullet$ for all $t \in T$, where, for $x \in (P \cup T)$, ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. For $X \subseteq (P \cup T)$, ${}^\bullet X$ and $X^\bullet$ are defined elementwise.

A *marking* of N is a multi set $M : P \to \mathbb{N}$. A transition t is *enabled under* M in N ($M \to^t$) iff ${}^\bullet t \leq M$ [1]. In this case, $M' = M - {}^\bullet t + t^\bullet$ is called *successor marking* of M w.r.t. t ($M \to^t M'$). A set U of transitions is *enabled under* M in N ($M \to^U$) iff $(\bigcup_{t \in U} {}^\bullet t) \leq M$. In this case the successor marking is given by $M' = M - (\bigcup_{t \in U} {}^\bullet t) + (\bigcup_{t \in U} t^\bullet)$.

A *net system* or *P/T-net* or *marked net* (N, M_0) consists of a net and an (initial) marking. A marking M is *reachable* in (N, M_0) if, for some $n \in \mathbb{N}$, there are $t_1, \ldots, t_n \in T$ and markings $M_1, \ldots, M_n$ such that $M_n = M$ and $M_0 \to^{t_1} M_1 \ldots \to^{t_n} M_n$. The set of reachable markings is denoted by $Reach(N, M_0)$. N is *finite* iff $P \cup T$ is finite and *safe* iff $M(p) \in \{0, 1\}$ for all $p \in P$ holds for any reachable marking M. For a safe net N, marking M is written as a subset $M \subseteq P$. The firing rule is reformulated accordingly. During the rest of the paper, only safe nets are considered.

Nodes $x_1, x_2 \in (P \cup T)$ of a net N are *in conflict* ($x_1 \# x_2$) iff there exist distinct transitions $t_1, t_2 \in T$ such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ and $(t_1, x_1), (t_2, x_2) \in F^*$, the reflexive and transitive closure of F. $x \in (P \cup T)$ is in *self-conflict* iff $x \# x$.

2.2 Branching Processes

The maximal branching process associates a partial order semantics to each safe net P/T-net N [9]. A *causal process* of a safe P/T-net N describes a possible run of N, displaying the causal dependencies of the events that take place during the run. A *branching process* can represent several alternative runs of N in one structure and hence may be seen as the union of some causal processes. It consists of an occurrence net and a homomorphism.

An *occurrence net* $ON = (B, E, G)$ is an acyclic net such that $|{}^\bullet b| \leq 1$ for all $b \in B$, no event $e \in E$ is in self conflict, and for all $x \in (B \cup E)$, the set of elements $y \in (B \cup E)$ such that $(y, x) \in G^*$ is finite. The elements of B are called *conditions* and the elements of E are called *events*. $\leq$ denotes the partial order induced by G on $B \cup E$; $<$ denotes the corresponding strict partial order;

[1] Here and in the following sets are sometimes interpreted as special multi sets and operations on multi sets rather than on sets are applied.

$Min(ON)$ denotes the set of minimal elements of ON w.r.t. $\leq$. A *causal net* is an occurrence net which also satisfies $|b^\bullet| \leq 1$ for every condition b.

A *homomorphism* from an occurrence net ON to a P/T-net (N, M_0) is a mapping $\pi : (B \cup E) \to (P \cup T)$ such that $\pi(B) \subseteq P$ and $\pi(E) \subseteq T$; for all $e \in E$, the restriction of π to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet\pi(e)$; the restriction of π to $e^\bullet$ is a bijection between $e^\bullet$ and $\pi(e)^\bullet$; the restriction of π to $Min(ON)$ is a bijection between $Min(ON)$ and M_0; for all $e_1, e_2 \in E$ it holds that if ${}^\bullet e_1 = {}^\bullet e_2$ and $\pi(e_1) = \pi(e_2)$ then $e_1 = e_2$.

Let (N, M_0) be a P/T-net. A *branching process* of (N, M_0) is a pair $\beta = (ON, \pi)$, consisting of an occurrence net ON, and a homomorphism π from ON to (N, M_0). β is a *causal process* of (N, M_0) iff ON is a causal net. Let (N_1, π_1) and (N_2, π_2) be two branching processes of (N, M_0). (N_1, π_1) and (N_2, π_2) are *isomorphic* iff, for some bijective homomorphism $\pi : N_1 \to N_2$, $\pi_2 \circ \pi = \pi_1$.

For each P/T-net (N, M_0), there exists a unique (up to isomorphism) maximal (w.r.t. prefix ordering; cf. [8]) branching process β_m. This is called the *unfolding of* (N, M_0).

An initial part of a causal process may be represented uniquely by the set of events contained in that part: A *configuration* C of a process β is a downward closed conflict free set of events, i.e., a set $C \subseteq E$ such that $e \in C$ implies $e' \in C$ for all $e' \leq e$, and, for all $e, e' \in C$, $e \# e'$ does not hold. $\emptyset$ denotes the empty configuration and, for each event e, $[e] = \{e' \in E \mid e' \leq e\}$ denotes the configuration generated by e. Note, that each configuration C of a branching process $\beta = (ON, \pi)$ uniquely determines a causal process, having C as set of events and all conditions connected to elements of C in ON as set of conditions. Arcs and inscriptions are also inherited from ON. Hence, notions defined for (causal) processes may also be applied to configurations and vice versa.

A configuration C defines a unique marking, consisting of exactly all conditions of ON which are marked after occurrence of all events of C: A *co-set* is a set B' of conditions of an occurrence net ON such that, for all $b \neq b' \in B'$, neither $b < b'$ nor $b' < b$ nor $b \# b'$. A *cut* is a maximal co-set B' (w.r.t. to set inclusion). The cut of a finite configuration C is defined by $Cut(C) = (Min(ON) \cup C^\bullet) \setminus {}^\bullet C$. C defines a marking of ON by $Mark(C) = \pi(Cut(C))$.

We have the following

Proposition 1.

(i) If Co is a co-set in the branching process $\beta = (ON, \pi)$ and $x_1, x_2 \in Co$ then $\pi(x_1) \neq \pi(x_2)$.
(ii) A marking M is reachable in N iff the maximal branching process β_m contains a configuration C such that $M = Mark(C)$.

Figure 1 shows a causal branching process of the P/T-net underlying the time Petri net TN_1. Names of conditions and events are omitted; the image of a vertex x under π (also called *label of* x) is written beside it.

2.3 Temporal Logics for Safe P/T-Nets

In this section we introduce the branching time logic $\mathcal{L}$ for safe P/T-nets defined in [9]. Later on we will extend this logic to time Petri nets and show how the model checking algorithm of [9] can be applied. Properties of the current marking of a P/T-net (N, M_0) are expressed using *place assertions*. E.g., a formula '$s_1 \wedge s_2$' expresses the fact that places s_1 and s_2 are marked. '$\Diamond\phi$' (*"possibly ϕ"*) means that a marking satisfying 'ϕ' is reachable. The derived operator '$\Box\phi$' (*"always ϕ"*) signifies that 'ϕ' is satisfied at all reachable markings.

Let N be a net. The syntax of $\mathcal{L}$ over N is presented given by the following grammar:

$$\phi ::= \mathtt{true} \mid s \in P_N \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \Diamond\phi.$$

Additionally, the following abbreviations are allowed:

$$\begin{array}{ll} \mathtt{false} & = \neg\mathtt{true} \\ (\phi_1 \vee \phi_2) & = \neg(\neg\phi_1 \wedge \neg\phi_2) \\ (\phi_1 \Rightarrow \phi_2) & = \neg\phi_1 \vee \phi_2 \\ \Box\phi & = \neg\Diamond\neg\phi \end{array}$$

Satisfaction of a formula ϕ w.r.t. a P/T-net (N, M_0) is defined inductively as follows. Let C be a finite configuration of a branching process β of (N, M_0). The superscript N is dropped if it is clear from the context.

$$\begin{array}{lll} (\beta, C) \models^N \mathtt{true} & & \\ (\beta, C) \models^N p & \text{iff } p \in \mathit{Mark}(C) & \\ (\beta, C) \models^N \neg\phi & \text{iff not } (\beta, C) \models^N \phi & \\ (\beta, C) \models^N \phi_1 \wedge \phi_2 & \text{iff } (\beta, C) \models^N \phi_1 \text{ and } (\beta, C) \models^N \phi_2 & \\ (\beta, C) \models^N \Diamond\phi & \text{iff } (\beta, C') \models^N \phi \text{ for some finite configuration } C' \supseteq C & \\ \beta \models^N \phi & \text{iff } (\beta, \emptyset) \models^N \phi & \\ N \models^N \phi & \text{iff } \beta_m \models^N \phi. & \end{array}$$

E.g., consider the following property Q: It always holds that, if p_1 and p_2 are marked, then possibly p_3 becomes marked. Q is expressed by the formula $\phi = \Box((p_1 \wedge p_2) \Rightarrow (\Diamond p_3))$, which is not satisfied by the P/T-net underlying TN_1.

2.4 McMillan-Unfoldings

The McMillan-unfolding of a P/T-net N was defined in [19] as a finite prefix β_f of the maximal branching process β_m of N such that each reachable marking is represented in β_f and, moreover, β_f contains an event labeled t for any transition t that can occur in N. A branching process with these properties will be called *complete* or a *McMillan-unfolding* of N.

The construction of a McMillan-unfolding is based on the notion of a *cut-off event*:

Let $\prec$ denote the partial order defined on the finite configurations of a branching process β by

$$[e] \prec [e'] \text{ iff } |[e]| < |[e']|.$$

An event e of β is a *cut-off event* (w.r.t. $\prec$) iff there exists an event e' of β such that $[e'] \prec [e]$ and $Mark([e]) = Mark([e'])$. Based on $\prec$ a complete finite prefix $\beta_\prec$ of a net system N can be defined as follows. Let E_f be the set of events of β_m given by: $e \in E_f$ iff no event $e' < e$ is a cut-off event and define $\beta_f = (B_f, E_f, F_f, \pi_f)$, the prefix of β_m having E_f as set of events. Note, that if in addition $\prec$ is total, there are at most as many non-cut-off events as there are reachable markings in N, i.e. in the worst case, the size of a McMillan-unfolding is linear in the size of the reachability graph.

Let $Sat(\phi)_N$ and $Sat_f(\phi)_N$ denote, respectively, the set of finite configurations of the maximal branching process and the set of configurations of a McMillan-unfolding of the P/T-net N that satisfy ϕ. Since $Sat(\phi)_N = \emptyset$ implies (not $N \models \phi$) which is equivalent to $N \models \neg\phi$, the model checking problem reduces to checking emptiness of $Sat(\phi)_N$. Esparza's theorem [9] states that for the latter it is sufficient to inspect $Sat_f(\phi)_N$:

Theorem 1. *Let N be a P/T-net and ϕ a formula in $\mathcal{L}$. Then:*

$$Sat(\phi)_N \neq \emptyset \text{ iff } Sat_f(\phi)_N \neq \emptyset.$$

In [10] an improved algorithm for computing a complete finite prefix of a P/T-net N is presented. All results presented in this paper also apply to the improved algorithm.

2.5 Time Petri Nets

Time Petri nets were introduced in [21] and further investigated e.g. in [24,26]. A *time Petri net TN* consists of a P/T-net together with an additional inscription of the transitions by closed time intervals with nonnegative rational bounds: $TN = (P, T, F, M_0, \chi)$ such that $\chi : T \to \mathcal{T}$, where $\mathcal{T} = \mathbb{Q}_{\geq 0} \times (\mathbb{Q}_{\geq 0} \cup \{\infty\})$. For $\chi(t) = (\mathit{eft}(t), \mathit{lft}(t))$ we call $\mathit{eft}(t)$ and $\mathit{lft}(t)$ the *earliest firing time* and, respectively, the *latest firing time* of t. The intended meaning is that $\mathit{eft}(t)$ and $\mathit{lft}(t)$ denote the minimal and, respectively, the maximal amount of time which may pass between enabling and successive occurrence of t. A time Petri net TN is called safe if its underlying P/T-net N is safe.

A *state* (M, I) of a time Petri net is given by a marking M of the underlying P/T-net together with a clock vector $I : T \to \mathbb{R}_{\geq 0}$. The clock vector associates a clock to each transition, showing the amount of time that has elapsed since the last enabling of the transition. State (M, I) is called *consistent* iff $I(t) \leq \mathit{lft}(t)$ for all $t \in \mathit{Enabled}(M)$.

For a clock vector I and a time delay $\theta \in \mathbb{R}_{\geq 0}$ such that $I(t) + \theta \leq \mathit{lft}(t)$ for all $t \in \mathit{Enabled}(M)$, $(I + \theta)$ is defined by $(I + \theta)(t) = I(t) + \theta$ for all $t \in T$. The *initial state* (M_0, I_0) is given by the initial marking M_0 of the underlying P/T-net and the *initial clock vector* I_0 defined by $I_0(t) = 0$ for all $t \in T$.

Two types of events are considered for time Petri nets, namely events for passing of time (*time-events*) and events for occurring of transitions (*occur-events*):

1. *Time-events*: A time-event $\theta \in \mathbb{R}_{\geq 0}$ is fireable at state (M, I) iff no transition is forced to occur in between, i.e. iff for all $t \in \mathit{Enabled}(M)$, $I(t)+\theta \leq \mathit{lft}(t)$. In this case, the successor state (M', I') is given by $M' = M$ and $I' = (I+\theta)$. Time-event θ is denoted by $(M, I) \rightarrow^{\theta} (M', I')$.
2. *Occur-events*: An occur-event $t \in T$ is fireable at state (M, I) iff
   - $t \in \mathit{Enabled}(M)$ and
   - $\mathit{eft}(t) \leq I(t) \leq \mathit{lft}(t)$.

   In this case, the successor state (M', I') w.r.t. (M, I) and t is given by $M' = (M \setminus {}^{\bullet}t) \cup t^{\bullet}$ and for all $t' \in T$

$$I'(t') = \begin{cases} 0 & \text{if } ({}^{\bullet}t' \cap {}^{\bullet}t) \neq \emptyset \\ I(t') & \text{otherwise.} \end{cases}$$

   Occur-event t is denoted by $(M, I) \rightarrow^{t} (M', I')$.

An *occur-step* consists of a nonempty set of concurrently fireable occur-events. A set $\{t_1, \ldots, t_n\} \subseteq T$ of occur-events is *concurrently fireable* at state (M, I) iff, for $1 \leq i \leq n$, t_i is fireable at state (M, I) and, for all $1 \leq i < j \leq n$, ${}^{\bullet}t_i \cap {}^{\bullet}t_j = \emptyset$. The successor state (M', I') is given by element wise application and the occur-step is denoted by $(M, I) \rightarrow^{\{t_1, \ldots, t_n\}} (M', I')$.

Note, that concurrent firing of a time-event and an occur-event is impossible, as is concurrent firing of two different time-events. To unify notation, we will sometimes denote time-events as *time-steps*.

A *firing schedule* of a time Petri net TN is a finite or infinite sequence $\sigma = (\overline{t_0}\ \overline{t_1}\ \overline{t_2}\ \ldots)$ such that, for all i, $\overline{t_i}$ is either an occur-step or a time-step. σ is called

- *fireable* from state $S = S'_0$ iff there exist states $S'_1, S'_2, S'_3, \ldots$ such that $\overline{t_i}$ is fireable at S'_i.
- *integral* iff all time steps occurring in σ are integral, i.e. iff $\overline{t_i} \in \mathbb{R}_{\geq 0}$ implies $\overline{t_i} \in \mathbb{N}$ for all i.

State S is called *reachable* from S_0 via σ iff some fireable firing schedule σ leads from S_0 to S. $\mathit{Reach}(TN)$ denotes the set of reachable states. A marking M is *reachable via* σ iff, for some clock vector I, there is a state (M, I) reachable via σ.

Based on the following theorem of Popova-Zeugmann [24] only time Petri nets with integral earliest and latest firing times and only time steps of length 1 – denoted by $\surd$ – are considered throughout the rest of the paper.

Theorem 2 (Popova-Zeugmann). *For each time Petri net TN there is a time Petri net TN' such that*

1. $\mathit{eft}(t) \in \mathbb{N}$ *and* $\mathit{lft}(t) \in (\mathbb{N} \cup \{\infty\})$ *for all* $t \in T$.

2. *A marking M is reachable in TN iff it is reachable via some integral firing schedule σ in TN'.*

Consider the net TN_1 of Fig. 1. The initial clock vector is given by $I_0 = (0,0,0,0)$. For example, the firing schedule $\sigma = (t_2\ \surd\ t_3\ \surd\ t_1\ t_2\ \surd\ t_3)$ is fireable at (M_0, I_0).

The behaviour of a time Petri net TN may be described by the reachability graph [24]. Note that this has the same property as for P/T-nets of not reflecting the concurrent behaviour, but instead representing all interleavings of concurrent events. To describe the concurrent behaviour of time Petri nets, in this section we adapt the notion of a branching process as well as some related notions to time Petri nets.

Definition 1 (Branching Process of a Time Petri Net).

(i) A *homomorphism* from an occurrence net $ON = (B, E, G)$ to a (safe) time Petri net $TN = (P, T, F, M_0, \chi)$ is a mapping

$$\pi : (B \cup E) \to (P \cup T \cup \{\surd\})$$

such that
1. $\pi(B) \subseteq P$ and $\pi(E) \subseteq (T \cup \{\surd\})$.
2. for all $e \in E$ the following holds:
   - If $\pi(e) \in T$, then the restriction of π to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet\pi(e)$ and the restriction of π to $e^\bullet$ is a bijection between $e^\bullet$ and $\pi(e)^\bullet$.
   - If $\pi(e) = \surd$, then ${}^\bullet e$ and $e^\bullet$ are cuts of ON and $\pi({}^\bullet e) = \pi(e^\bullet)$.
3. the restriction of π to $Min(ON)$ is a bijection between $Min(ON)$ and M_0.
4. for all $e_1, e_2 \in E$, if ${}^\bullet e_1 = {}^\bullet e_2$ and $\pi(e_1) = \pi(e_2)$ then $e_1 = e_2$.

(ii) A *branching process* of a time Petri net N is a pair $\beta = (ON, \pi)$, consisting of an occurrence net ON and a homomorphism π from ON to TN. β is a *causal process* of N iff ON is a causal net.

Branching process $\beta = (B, E, G, \pi)$ of TN defines a set of transition sequences of TN: Let $e_1 \dots e_n$ be a sequence of events of β such that, for $1 \le i < n$, $(e_i, e_{i+1}) \in G \circ G$. Then $\pi(e_1) \dots \pi(e_n)$ is a transition sequence in N. $OC(\beta)$ denotes the set of these sequences.

Definition 2. *Let TN be a time Petri net and β a branching process of TN. β is called admissible iff every $\sigma \in OC(\beta)$ is a firing sequence of TN.*

The following theorem may be proven by adapting the proof of Engelfriet's corresponding theorem for P/T-nets:

Theorem 3. *For each (finite) time Petri net TN, there exists a unique (up to renaming of conditions and events) admissible branching process β_m called unfolding of TN which is maximal w.r.t. the prefix ordering on the set of all admissible branching processes of TN.*

The notions of a configuration and of a base configuration are defined as for P/T-nets, but w.r.t. admissible processes.

Definition 3. *Let TN be a time Petri net and $\beta = (B, E, G, \pi)$ an admissible branching process of TN.*

(i) A configuration C of β is a downward closed conflict free set of events, i.e., a set $C \subseteq E$ such that $e \in C$ implies $e' \in C$ for all $e' \leq e$, and, for all $e, e' \in C$, $e \# e'$ does not hold. The empty configuration is denoted by $\emptyset$.

(ii) For any event $e \in E$, the basic configuration of e is given by

$$[e] = \{e' \in E \mid e' \leq e\}.$$

The notions of cut and of the marking of a configuration carry over from P/T-nets directly. A new notion relevant for time Petri nets is that of the state of a configuration.

Definition 4 (State of a Configuration). *Let TN be a time Petri net, $\beta = (B, E, G, \pi)$ an admissible branching process of TN, and C a configuration of β.*

(i) The clock vector *$Clocks(C) : T \to \mathbb{N}$ of C is defined inductively as follows:*
Base:

$$Clocks(\emptyset)(t) = 0$$

Step: *Let $C = C' + \{e\}$ for some configuration C' of β and some $e \in E$.*
Case 1: $\pi(e) \in T$. Then:

$$Clocks(C)(t) = \begin{cases} Clocks(C')(t) & \text{if } t \in (Enabled(Mark(C)) \\ & \cap Enabled(Mark(C'))) \text{ and} \\ & {}^\bullet t \cap {}^\bullet \pi(e) = \emptyset \\ 0 & \text{otherwise, } t \in Enabled(Mark(C)) \end{cases}$$

Case 2: $\pi(e) = \surd$. Then:

$$Clocks(C)(t) = \begin{cases} Clocks(C')(t) + 1 & \text{if } t \in Enabled(Mark(C)) \\ Clocks(C')(t) & \text{otherwise.} \end{cases}$$

(ii) The state of C is defined as $State(C) = (Mark(C), Clocks(C))$.

A different notion of the change of time is possible when storing time with tokens (resp. conditions). This way it becomes easier to keep the information within the branching process.

Definition 5 (Age of a Token). *Let TN be a time Petri net, $\beta = (B, E, G, \pi)$ an admissible branching process of TN and C a configuration of β.*
The age of a token $Age : B \to \mathbb{N}$ is defined as follows:

$$Age(b) = \begin{cases} 0 & \text{if } \pi({}^\bullet(b)) \neq \surd \\ Age(b' + 1) & \text{otherwise, with } b' \in {}^\bullet({}^\bullet(b)) \text{ and } \pi(b') = \pi(b) \end{cases}$$

With this, one can define a state function

$$State'(C) = (Mark(C), Age(Cut(C))).$$

Lemma 1. *Let TN be a time Petri net, $\beta = (B, E, G, \pi)$ an admissible branching process of TN and C a configuration of β.*

$$Clocks(C)(t) = \begin{cases} \bot & \text{if } t \notin Enabled(Mark(C)) \\ \min_{b \in {}^\bullet(t)}(Age(b)) & \text{otherwise} \end{cases}$$

The logic $\mathcal{L}$ is now extended to time Petri nets. The syntax of formulae will not be changed. The satisfaction relation is defined as follows:

Definition 6. *Let TN be a time Petri net and ϕ a formula over TN, i.e. over the underlying P/T-net N. Satisfaction of a formula ϕ w.r.t. a time Petri net TN is defined inductively as follows. Let β denote an admissible branching process of TN and C, C' finite configurations of β. The superscript TN may be dropped.*

$$\begin{array}{lll}
(\beta, C) \models_t^{TN} \texttt{true} & & \\
(\beta, C) \models_t^{TN} p & \text{iff } p \in Mark(C) & \\
(\beta, C) \models_t^{TN} \neg\phi & \text{iff not } (\beta, C) \models_t^{TN} \phi & \\
(\beta, C) \models_t^{TN} \phi_1 \wedge \phi_2 & \text{iff } (\beta, C) \models_t^{TN} \phi_1 \text{ and } (\beta, C) \models_t^{TN} \phi_2 & \\
(\beta, C) \models_t^{TN} \Diamond\phi & \text{iff } (\beta, C') \models_t^{TN} \phi \text{ for some finite configuration } C' \supseteq C & \\
\beta \models_t^{TN} \phi & \text{iff } (\beta, \emptyset) \models_t^{TN} \phi & \\
TN \models_t^{TN} \phi & \text{iff } \beta_m \models_t^{TN} \phi. &
\end{array}$$

As an example, property Q of the introductory example is expressed by the formula $\phi = \Box((p_3 \wedge p_4) \Rightarrow (\Diamond p_1))$, which is not satisfied by TN_1.

Suppose that we are given a time Petri net TN together with an $\mathcal{L}$-formula ϕ over TN and that we have to check whether TN satisfies ϕ. ¿From the considerations of Sect. 1 it follows that it is not sufficient to check ϕ against the intersection of the maximal branching process β_m of TN and the McMillan-unfolding $McM(N)$ of the underlying P/T-net N. The reason for this is that the definition of a cut-off event does not consider the clock vector part of states of TN. This will be done by the following definition.

Definition 7 (Time Cut-Off Event). *Let $\beta = (B, E, G, \pi)$ be an admissible process of a time Petri net TN and $\prec$ the partial order on configurations defined by $[e] \prec [e']$ iff $|[e]| < |[e']|$. An event $e \in E$ is called (time) cut-off event iff there is some event $e' \in E$ such that*

$$[e'] \prec [e] \text{ and } State([e']) = State([e]).$$

In the next section we will define the time expansion $X(TN)$ of TN as a safe finite P/T-net embodying the clock vector as part of the structure and its possible changes as part of the behaviour. It will then turn out that the McMillan-unfolding $McM(X(TN))$ of the time expansion is sufficient for checking satisfiability of the formula ϕ.

3 Bisimulation of Time Petri Nets and P/T-Nets

In Sect. 4, to each time Petri net TN an 'equivalent' ordinary P/T-net N is associated as its time expansion. Intuitively, TN and N are called equivalent, if both can perform the 'same' set of sequences of (concurrent) steps. In this section, the intuitive relation between a time Petri net and its time expansion is formalized by the notion of place distributive step-bisimulation equivalence. Based on this notion, it is shown that bisimular nets essentially satisfy the same sets of temporal logic formulae.

Definition 8 (Step-Bisimulation). Let TN be a time Petri net and N a P/T-net. Let $\sim_r \subseteq Reach(TN) \times Reach(N)$ and $\sim_t \subseteq (T_{TN} \cup \{\surd\}) \times T_N$.

(i) $(\sim_r, \sim_t)$ is called *bisimulation from TN to N* iff
1. $\sim_t^{-1}$ is a mapping from T_N onto $(T_{TN} \cup \{\surd\})$.
2. $S_{0TN} \sim_r M_{0N}$.
3. If $S \rightarrow^{\{t_1,\ldots,t_n\}} S'$ in TN and $S \sim_r \overline{M}$ for some marking $\overline{M}$ of N, then there exist transitions $t'_1, \ldots, t'_n$ of N and a marking $\overline{M'}$ of N such that
   (a) $t_i \sim_t t'_i$ for $1 \leq i \leq n$,
   (b) $S' \sim_r \overline{M'}$, and
   (c) $\overline{M} \rightarrow^{\{t'_1,\ldots,t'_n\}} \overline{M'}$ in N.
4. If $\overline{M} \rightarrow^{\{t'_1,\ldots,t'_n\}} \overline{M'}$ in N and $S \sim_r \overline{M}$ for some state S of TN, then there exist transitions $t_1, \ldots, t_n$ of TN and a state S' of TN such that
   (a) $t_i \sim_t t'_i$ for $1 \leq i \leq n$,
   (b) $S' \sim_r \overline{M'}$, and
   (c) $S \rightarrow^{\{t'_1,\ldots,t'_n\}} S'$ in TN.

(ii) A bisimulation $(\sim_r, \sim_t)$ is called *concurrency preserving* or *step-bisimulation from TN to N* iff, for all $t'_1, t'_2 \in T_N$ it holds that

$$\text{If } t'_1 \neq t'_2 \text{ and } \overline{M} \rightarrow^{\{t'_1,t'_2\}} \text{ in } N \text{ and } t_i \sim_t t'_i, i = 1, 2, \text{ then } t_1 \neq t_2.$$

(iii) A bisimulation $(\sim_r, \sim_t)$ is called *place-distributive* (*pd* for short) iff, for all $p \in P_{TN}$, there are $k \in \mathbb{N}$ and $p^1, \ldots, p^k \in P_N$ such that
- $(M \cup \{p\}, I) \sim_r \overline{M}$ implies $|\overline{M} \cap \{p^1, \ldots, p^k\}| = 1$ and
- $(M - \{p\}, I) \sim_r \overline{M}$ implies $\overline{M} \cap \{p^1, \ldots, p^k\} = \emptyset$.

For a pd bisimulation $(\sim_r, \sim_t)$ we write $p \sim_r p'$ iff $p' \in \{p^1, \ldots, p^k\}$ with $p^1, \ldots, p^k$ as above.

Proposition 2. *Let N and TN as above and $(\sim_r, \sim_t)$ a step-bisimulation from TN to N. If $(M, I) \rightarrow^\sigma (M', I')$ for a step sequence σ in TN and $(M, I) \sim_r \overline{M}$ then there exist $\overline{M'}$ and a step sequence $\overline{\sigma}$ in N such that $(M', I') \sim_r \overline{M'}$, $\sigma \sim_t \overline{\sigma}$[2] and $\overline{M} \rightarrow^{\overline{\sigma}} \overline{M'}$.*

Definition 9 (Place Expansion).
Let N and TN as above and $(\sim_r, \sim_t)$ a pd step-bisimulation from TN to N. The *place expansion* $PX(\phi)$ of a formula ϕ over TN is given by

[2] Application of a relation to a set or a sequence of elements is understood pointwise.

1. $PX(\texttt{true}) = \texttt{true}$
2. $PX(p) = (p^1 \vee \ldots \vee p^k)$ for all $p \in P$, where $\{p^1, \ldots, p^k\} = \{p' \mid p \sim_r p'\}$.
3. $PX(\neg\phi) = \neg(PX(\phi))$.
4. $PX(\phi_1 \wedge \phi_2) = PX(\phi_1) \wedge PX(\phi_2)$.
5. $PX(\Diamond\phi) = \Diamond(PX(\phi))$.

Lemma 2. *Let $(\sim_r, \sim_t)$ a pd step-bisimulation from TN to N. Let $(M, I) \sim_r \overline{M}$ and ϕ a formula over TN. Then:*

$$(M, I) \models_t \phi \text{ iff } \overline{M} \models PX(\phi)$$

Proof. The proof is done by induction on the structure of ϕ.

Theorem 4. *Let TN be a time Petri net, N a P/T-net, $(\sim_r, \sim_t)$ a place-distributive step-bisimulation from TN to N, and ϕ a formula over TN. Then*

$$TN \models_t \phi \text{ iff } N \models PX(\phi).$$

4 Expanding Time Restrictions

The key idea of the approach presented here consists of transforming time restrictions into net structure, i.e. representing them by additional places, transitions, and arcs. Following this approach, for a given time Petri net TN the time-expansion $EX(TN)$ is constructed as an ordinary P/T-net. Later on, it will be shown that a formula ϕ is satisfied by TN iff it is satisfied by the time-expansion $EX(TN)$. Subsequently, the McMillan-unfolding $McM(TN)$ of TN is defined by (an abstraction of) $McM(EX(TN))$. As it turns out, for the actual computation of $McM(TN)$ the explicit construction of the (complex) net $EX(TN)$ is not necessary.

The new notion of time-expansion presented here improves the method of [6] in several aspects:

- The restriction of *divergent time* (cf. [6]) is removed.
- Transitions may have an infinite upper time bound.
- TN and $EX(TN)$ are bisimular.
- The finite prefix of $EX(TN)$ may be computed directly and efficiently.

Consider a time Petri net TN. Its behaviour differs from the behaviour of the underlying P/T-net in that firing of transitions depends also on the current clock vector, and that time steps are possible. Hence both aspects have to be included in the time expansion.

Clock vectors will not directly be represented in the time expansion. Instead, for each place and each token we will keep track of the number of time units elapsing while the token resides in the place. This is done by providing appropriate instances of each place storing the token age. Fireability of a transition then depends on the minimal token age in its preset. Hence, for each transition, we

have to provide one copy for each instance of the preset at which the transition may fire.

In general, a token could reside in a place forever. Unfortunately this would lead to an infinite number of copies needed for such a place. This is avoided by giving a finite upper bound for each place such that we do not need to distinguish different token ages above this bound.

Following these considerations, a time step at a marking M is represented by a transition which increments the token age for each place marked under M. Here the following problem arises: If $M' \subseteq M$ in the time expansion, and the time step for M is enabled, then the time step for M' is also enabled. But occurrence of the time step for M' would increase the age of some tokens, leaving some tokens untouched. This would lead to a wrong time distribution. This problem is fixed by adding one more instance for each place indicating that the place is empty. Moreover, for each place which is not marked at M, this instance is added to both the preset and the postset of the time step. Hence, for each time step, exactly one instance of each place is contained in the preset as well as in the postset.

For the formal definition of time expansion we need some more notation. So let $TN = (P, T, F, M_0, \chi)$ be a time Petri net. Then:

- For each $p \in P$,

$$max(p) = \begin{cases} 0 & \text{if } p^\bullet = \emptyset \\ max(\{eft(t) \mid t \in p^\bullet\}) & \text{otherwise and} \\ & lft(t) = \infty \text{ for all } t \in p^\bullet \\ max(\{lft(t) \mid t \in p^\bullet \wedge lft(t) \neq \infty\}) & \text{otherwise.} \end{cases}$$

- Let $Q \subseteq P$. Then $\mu : P \to (\mathbb{N} \cup \{\perp\})$ is called *Q-instance* iff
  - $\mu(p) \in \mathbb{N}$ iff $p \in Q$ and
  - $\mu(p) \leq max(p)$

  for all $p \in P$.
- Let $Q \neq \emptyset$ and μ a Q-instance. Then $min(\mu) = min\{\mu(p) \mid \mu(p) \in \mathbb{N}\}$.
- For a Q-instance μ we define the Q-instance $\mu \oplus 1$ by

$$(\mu \oplus 1)(p) = \begin{cases} \perp & \text{if } \mu(p) = \perp \\ max(p) & \text{if } \mu(p) = max(p) \\ \mu(p) + 1 & \text{otherwise.} \end{cases}$$

Definition 10 (Time-Expansion).
Let $TN = (P, T, F, M_0, \chi)$ be a time Petri net. The *time-expansion* of TN is given by the P/T-net

$$EX(TN) = (P_{EX}, T_{EX}, F_{EX}, M_{EX_0})$$

defined as follows:

- $P_{EX} = \bigcup_{p \in P} \{p^\perp, p^0, \ldots, p^{max(p)}\}$.

- $T_{EX} = \{t^\mu \mid t \in T \text{ and } \mu \text{ is a } {}^\bullet t\text{-instance s.t. } min(\mu) \geq eft(t)\}$
  $\cup \{\surd^\mu \mid \mu(p) < max(p) \text{ for all } p \in P\}$.
- F_{EX} is given by
  - ${}^\bullet(t^\mu) = \{p^{\mu(p)} \mid p \in {}^\bullet t\}$
  - $(t^\mu)^\bullet = \{p^0 \mid p \in t^\bullet\}$
  - ${}^\bullet(\surd^\mu) = \{p^{\mu(p)} \mid p \in P\}$
  - $(\surd^\mu)^\bullet = \{p^{\mu(p)\oplus 1} \mid p \in P\}$.
- $M_{0_{EX}} = \{p^0 \mid p \in M_0\} \cup \{p^\perp \mid p \notin M_0\}$.

Theorem 5. *Let TN be a time Petri net and $EX(TN)$ its time expansion. Then TN and $EX(TN)$ are place-distributive step-bisimular.*

Corollary 1. *Let TN be a time Petri net and ϕ a formula over TN. Then:*

$$TN \models_t \phi \text{ iff } EX(TN) \models PX(\phi) \text{ iff } McM(EX(TN)) \models PX(\phi).$$

5 Computation of McMillan-Unfoldings

Now we have reduced the model checking problem of a time Petri net TN to the model checking problem of the P/T-net $EX(TN)$, but at the expense of a larger formula. Since each reachable marking of $EX(TN)$ contains exactly one instance of each place p of TN, the same holds true for each cut of the maximal branching process of $EX(TN)$. Hence, we may as well remove the $\perp$-instances of places, forget about the 'labels' of the remaining places, and check the resulting process against the original formula:

Definition 11 (Reduced Branching Process). Let TN be a time Petri net, $EX(TN)$ its time expansion, and β a process of $EX(TN)$. The *reduction* $Red(\beta)$ *of* β is constructed from β by

- removing all conditions labeled $p^\perp$ for some $p \in P_{TN}$ and
- replacing
  - each label of the form p^n for some $p \in P_{TN}$ by p,
  - each label of the form t^μ for some $t \in T_{TN}$ by t, and
  - each label of the form $\surd^\mu$ by $\surd$.

Theorem 6. *Let TN be a time Petri net, $EX(TN)$ its time expansion, and β a process of $EX(TN)$. Then*

$$McM(EX(TN)) \models PX(\phi) \text{ iff } Red(McM(EX(TN))) \models \phi.$$

Hence, it is reasonable to define the McMillan-unfolding of a time Petri net in the following way:

Definition 12 (McMillan-Unfolding). Let TN be a time Petri net. The McMillan-unfolding $McM(TN)$ of TN is defined by

$$McM(TN) = Red(McM(EX(TN))).$$

Corollary 2. *The McMillan-unfolding $McM(TN)$ of a time Petri net TN is sufficient for model checking of formulae over TN.*

Corollary 3. *The finite prefix may also be computed directly using the $State'$ function. Given the original algorithm using the cut-off condition*

$$[e'] \prec [e] \text{ and } State'([e']) = State'([e]).$$

Remark 1. Starting from the order $\prec$ defined in [10] we are able to define an adequate total order $\prec_t$ for time Petri nets. As a consequence, we are able to show that the size of $\beta_{\prec_t}$ is linearly bounded in the number of reachable states of TN.

6 Experimental Results

This section provides the results of the algorithm on some small but yet complete examples. The algorithm has been implemented as an extension of the already existing model checker from the PEP tool[23]. This implementation is available from the Internet, but does not yet offer the improved total order from [10]. Nevertheless, it proves the proposed theory to be feasible and useable on a range of problems. All computations were made on a dual processors Linux system with 2x450 MHz and a total of 1GByte of RAM.

The first example modelled with time Petri nets is a simple version of the alternating bit protocol. It is a well-known transmission protocol consisting of a server that sends messages and a client who receives them replying an acknowledge message in case of success. Messages are sent via lossy channel so data and ack packets may be lost. In both cases the message is resent after some time out. Time intervals are set for all actions concerning the channel, i.e. sending and receiving, and also for some delays during transmission as we have an asynchronous protocol. The size of the net and its unfolding are given in tab. 2 together with its unfolding time and deadlock checking times in seconds.

Places	Transitions	Conditions	Events	Unfolding (s)	Checking(s)
25	25	2946	932	.2	.3

Fig. 2. Alternating bit protocol

The second example given is an implementation of Fischer's protocol ([1]). It is a very basic mutual exclusion protocol only allowing one process to reach the criticial section and then stopping, though it can be extended for an arbitrary number of rounds. The protocol itself is very simple and scalable for any number of processes. It consists of a global variable and special timing parameter on every round of the algorithm. Provided the interval for writing ends before the one for reading starts the protocol guarantees letting only one process enter the critical

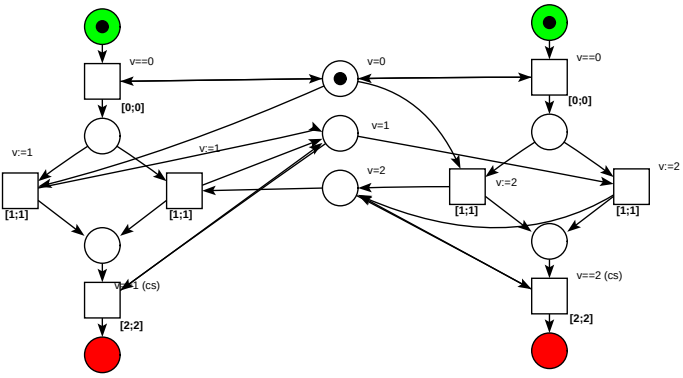


Fig. 3. Fischer's protocol for 2 processes

section. Due to its simplicity and scalability this protocol is likely to be used as a benchmark for timed systems (cf. [17]). Figure 3 shows a Petri net model for 2 processes and a global variable v. A process enters its critical section after firing the last transition with attached *cs*. The other transitions have inscriptions stating their influence on v.

As shown in Table 4 the size of the unfolding grows exponentially with the number of processes. This limits the scope of the proposed algorithm, though the table also shows its benefits. The time for checking of the mutual exclusion property is done in almost no time and does not change much with the growing unfolding. So the crucial part for further investigations will be the efficiency of the unfolder. As the current implementation does not include the proposed technique from [10] and has not been optimized in its implementation, the algorithm itself may yield some promising improvements, besides e.g. parallel programming benefits like [15].

Processes	Places	Transitions	Conditions	Events	Unfolding (s)	Checking(s)
2	11	8	127	47	<.1	<.1
3	16	15	3616	1080	0.6	.1
4	21	24	245025	63451	4572.9	.4

Fig. 4. Fischer's protocol

7 Related Work

Approaches to model checking of time Petri nets w.r.t. (timed) temporal logics are either based on the timed state graph (TSG) [28,29], or on the timed branching process [3,18,28] of a time Petri net.

In [29], model checking of safe time Petri nets with rational firing bounds w.r.t. a real time extension of linear time temporal logic is considered, which is based on differences between timing variables rather than absolute time points. A finite graph representing the state space is computed whose nodes consist of reachable markings augmented by additional sets of inequalities describing timing conditions. Since each reachable marking is directly represented by at least one node, this approach faces the state explosion problem.

The time state graph construction of [29] is improved in [18] in several ways. First, a coarser semantics which does not take into account implicit ordering of events is considered. Second, the finite prefix of the underlying net is used to speed-up searching the time state graph.

Model checking against TCTL is explored in [28]. Here, a region graph construction is used to compute a finite representation of the timed state space, cf. the construction for timed automata in [2].

A notion of timed process of a time Petri net, denoting a causal process which is realizable under the time constraints of the net, is introduced in [3]. A method is developed to compute all valid timings for a causal process of a time Petri net with divergent time.

In [25], the class of safe time Petri nets is restricted to *time independent choice* nets and analysed w.r.t. reachability. The reachability checking algorithm is based on a finite prefix of the underlying net. Like in [29], timing conditions are represented explicitely by sets of inequalities as additional inscriptions of events. Using the prefix construction, an asynchronous circuit with bounded delay is verified w.r.t. hazard-freedom.

Building a complete finite prefix is still a bottleneck in partial order verifications. Although [10] gave a major improvement in terms of the size of the unfolding, the construction is still too slow for large systems. Some new algorithms try to cope with this. In [16] the search for possible new events in the unfolding is changed. With information from the structure of the net it is possible to reduce the number of calls to this search. Furthermore the search may be improved if transition's presets are kept in a tree structure to avoid redundancy.

In [14] the cutoff-criterion is refined. With the use of logic programs it is shown that a much smaller prefix may suffice to check reachability and deadlock properties. The approach was implemented as a minimizer after actual unfolding but may also be integrated into the unfolding algorithm. As the question if this minimized prefices are useable for model checking is left open, its benefits for a general unfolder are harder to predict.

The approach presented in this paper may also be compared to timed automata (cf. e.g. [2]). E.g. checking reachability in a time Petri net TN can be reduced to reachability analysis in a timed automaton TA in the following way: Given a safe time Petri net TN, first compute its reachability graph $G = (V, E)$, giving the states and the transitions of TA. Introduce a clock c_t for each transition t of TN. The inscription of a transition $M \xrightarrow{t} M' \in E$ consists of the guard $\mathit{eft}(t) \leq c_1 \leq \mathit{lft}(t)$ and the set of clocks to be reset is given by $\{t' \mid t' \in \mathit{Enabled}(M - {}^\bullet t)\}$. Moreover, for each state M, the invariant is given

by `true`. Unfortunately, this construction is very expensive: First, the size of G may be exponential in the size of TN; second, in order to check reachability in TA, the region graph construction is needed, bringing along an additional exponential blow up.

On the other hand, simulation of general timed automata by time Petri nets seems to be unfeasible.

8 Conclusion

We have defined the finite prefix of a time Petri net TN, based on the time-expansion of TN. By showing that this is sufficient for applying Esparza's model checking algorithm for safe P/T-nets, we have established a first – but still substantial – step towards quantitative analysis of distributed real-time systems based on partial order semantics. The next steps will consist in extending our approach to *real-time temporal logic*.

Acknowledgements

The authors would like to thank the members of the BAT/Porta group for helpful comments and discussions on the paper.

References

1. M. Abadi and L. Lamport: An Old-Fashioned Recipe for Real-Time. In *Real Time: Theory in Practice*. Volume 600 of LNCS. Springer-Verlag (1992) 1–27 177
2. R. Alur, C. Courcoubetis, and D. Dill: Model Checking for Real-time Systems. In *Proc. 5th Symp. LICS*. (1990) 414–425 163, 179
3. T. Aura and J. Lilius: A causal semantics for time Petri nets. In *Theoretical Computer Science*, Vol 243, No. 1–2 (2000) 409–447 178, 179
4. B. Berthomieu, M. Diaz: Modelling and Verification of Time Dependent Systems Using Time Petri Nets. In *IEEE Transactions on Software Engineering*, Volume 17/3 (1991) 259–273
5. E. Best: Partial Order Verification with PEP. In G. Holzmann, D. Peled, and V. Pratt (eds.): *Proc. of POMIV'96*, American Mathematical Society (1996) 305–328 163, 164
6. B. Bieber and H. Fleischhack: Model Checking of Time Petri Nets Based on Partial Order Semantics. In *Proceedings of CONCUR'99*. Volume 1664 of LNCS. Springer-Verlag (1999) 210–225 163, 174
7. E. M. Clarke, E. A. Emerson, and A. P. Sistla: Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS* 8 (1986) 244–263 163
8. J. Engelfriet: Branching Processes of Petri Nets. *Acta Informatica* **28**. (1991) 575–591 166
9. J. Esparza: Model Checking Using Net Unfoldings. *Science of Computer Programming* **23**. Elsevier (1994) 151–195 163, 165, 167, 168

10. J. Esparza, S. Römer, and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. In *Proc. of TACAS'96*. Volume 1055 of LNCS. Springer-Verlag (1996) 87–106 168, 177, 178, 179
11. H. Fleischhack, and J. Tapken: An M-Net Semantics for a Real-Time Extension of μSDL,. In *Proc. FME'97*. Graz (1997).
12. P. Godefroid: Using partial orders to improve automatic verification methods. In *Proc. Workshop on Computer Aided Verification* (1990) 176–185 163
13. B. Graves: Computing Reachability Properties Hidden in Finite Net Unfoldings. In *Proc. of FST&TCS'97*. Volume 1346 of LNCS. Springer-Verlag (1997) 327–341
14. K. Heljanko: Minimizing Finite Complete Prefixes. In *Proceedings of CS&P'99*. Warsaw University (1999) 83–95 179
15. K. Heljanko, V. Khomenko and M. Koutny: Parallelisation of the Petri Net Unfolding Algorithm. To appear in *Proc. of TACAS'2002*. Grenoble, France (2002) 178
16. V. Khomenko, and M. Koutny: Towards An Efficient Algorithm for Unfolding Petri Nets. In *Proc. of CONCUR'2001*, K. G.Larsen, M.Nielsen (Eds.). Volume 2154 of LNCS. Springer-Verlag (2001) 366–380. 179
17. K. Larsen, P. Petterson and W. Yi: Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*. Volume 965 of LNCS. Springer-Verlag (1995) 62–88. 178
18. J. Lilius: Efficient State Space Search for Time Petri Nets. In *MFCS Workshop on Concurrency*. Volume 18 of ENTCS. Elsevier (1998). 178, 179
19. K. McMillan: *Symbolic Model Checking – an Approach to the State Explosion Problem*. PhD Thesis, SCS, Carnegie Mellon University (1992). 163, 167
20. K. L. McMillan: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. Workshop on Computer Aided Verification* Volume 663 of LNCS. Springer-Verlag (1992) 164–174 163
21. P. Merlin, D. Farber: Recoverability of Communication Protocols – Implication of a Theoretical Study. *IEEE Transactions on Software Communications* **24** (1976) 1036–1043 168
22. X. Nicollin, J. Sifakis, and S. Yovine: From ATP to Timed Graphs and Hybrid Systems. *Acta Informatica* **30**. Springer-Verlag (1993) 181–202 163
23. *PEP homepage* http://parsys.informatik.uni-oldenburg.de/~pep 164, 177
24. L. Popova: On Time Petri Nets. *Journal of Information Processing and Cybernetics*. EIK 27:4 (1991) 227–244. 168, 169, 170
25. A. Semenov, A. Yakovlev, and A. Koelmans: *Time Petri net unfoldings and hardware verification*. University of Newcastle (1998) (Draft) 179
26. P. Starke: *Analyse von Petri-Netz-Modellen*. Teubner Verlag, Stuttgart (1990) (in German) 168
27. C. Stehno: *Entfaltung von Zeit-Petrinetzen, Integration in das PEP-System und Optimierungen*. University of Oldenburg (2000) (Master thesis, in german)
28. I. Virbitskaite, E. Pokozy: *On Partial Order Verification of Time Petri Nets*. University of Novosibirsk (1998) (Draft) 178, 179
29. T. Yoneda, B.-H. Schlingloff: Efficient Verification of Parallel Real-Time Systems. *Journal of Formal Methods in System Design*, Vol. 11-2. (1997) 187–215 178, 179

Verification of a Revised WAP Wireless Transaction Protocol*

Steven Gordon[1], Lars Michael Kristensen[2,**], and Jonathan Billington[2]

[1] Institute for Telecommunications Research, University of South Australia
[2] Computer Systems Engineering Centre, University of South Australia
{Steven.Gordon,Lars.Kristensen,Jonathan.Billington}@unisa.edu.au

Abstract. The Wireless Transaction Protocol (WTP) is part of the Wireless Application Protocol (WAP) architecture and provides a reliable request–response service. The state space method of Coloured Petri Nets has been used to analyse a revised version of WTP, to gain a high level of confidence in the correctness of the design. Full state space analysis allows us to prove properties of the protocol for maximum values of the retransmission counters used in GSM networks (values are 4). However, the size of the state space grows rapidly as the maximum counter values are increased. We apply the sweep-line method to take advantage of the progress present in the protocol, notably the progression through major states of the protocol entities, and the increasing nature of the retransmission counters. The sweep-line method allows us to prove properties of the protocol for larger counter values, including those used in Internet Protocol (IP) networks (where the maximum values are 8). As a result, verification of WTP can be performed for the two most important networks (GSM and IP), the ones for which the WAP standard gives recommended maximum values for the retransmission counters.

1 Introduction

Coloured Petri nets (CPNs) [15] have been used extensively to model and analyse distributed systems [15,2]. Several factors have contributed to the popularity of CPNs, including the compact manner in which systems can be modelled, and the tool support in the form of Design/CPN [7]. The motivation for modelling and analysing the Wireless Transaction Protocol (WTP) [22] with CPNs is to gain a high level of confidence in the design of the protocol. This is achieved by following a protocol engineering methodology [3], of which the main step is to verify that the *protocol specification* is a faithful refinement of the *service specification*. For WTP, this begins by creating a CPN model of the Transaction Service, from which the possible set of sequences of service primitives is generated by viewing the state space as a Finite State Automaton (FSA). This is the Transaction Service language. State space analysis of the Transaction Protocol CPN allows

* The work of Lars M. Kristensen and Jonathan Billington has been supported by an Australian Research Council (ARC) Discovery Grant (DP0210524).
** Supported by the Danish Natural Science Research Council.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 182–202, 2002.
© Springer-Verlag Berlin Heidelberg 2002

several dynamic properties to be proved (e.g. absence of deadlocks, livelocks). The events in the protocol state space that do not represent service primitives are then abstracted out, and the resulting protocol language is compared to the Transaction Service language. The two must be language equivalent for the protocol to refine the service. We apply the FSM tool [1] to conduct the language comparison as there is no support for this in Design/CPN.

This paper has arisen from our research on the verification of WTP [9,10,11]. An initial CPN model and analysis results of the Transaction Service were published in [9]. This model has since been updated to enforce end-to-end behaviour of the service and allow abort primitives to be submitted after a successful transaction. Similarly, an initial CPN model of the Transaction Protocol was presented in [10], with analysis results revealing three errors in the protocol. Improvements have been made to produce a simpler and clearer model, with the main change being only one transaction is modelled. Analysis of this improved model identified further errors, and changes to the WTP Specification [22] are suggested in [11], leading to a *Revised Transaction Protocol*. The contribution of this paper is the verification and analysis of the Revised Transaction Protocol, including comparison with the Transaction Service. Details of the identified errors and suggested changes leading to the Revised Transaction Protocol are not the focus of this paper. The revised protocol and CPN models will be described only to the level of detail necessary to understand the analysis approach and results.

The state space analysis of the Transaction Protocol presents two practical challenges: the explosion in the number of states and the need to perform the analysis for different parameters. For the latter problem, our objectives are to prove the desired properties of the protocol for practical values of parameters, two of which are the recommended values of parameters in GSM networks and in networks using the Internet Protocol (IP). Several methods have been developed for alleviating the state explosion problem in CPNs, e.g. symmetries [15], equivalence classes [15], and stubborn sets [19,20] and used in various applications (e.g. [16,17]). In this paper we apply the recently developed sweep-line method [6,5] in the verification of WTP [22]. The sweep-line method exploits the progress present in a system to explore the full state space while storing only small fragments of the state space in memory at a time. As demonstrated by some initial case studies [6] this can lead to a significant reduction in peak memory usage. The sweep-line method is used for the values of the parameters of WTP, where full state spaces (state spaces in their most basic form) could not be generated with the available computing power. This paper describes how the sweep-line method is used, and discusses the relevance of the results for WTP, and for protocols in general.

The sweep-line method is related to the bit-state hashing [12] and state space caching [14] methods. For our verification purposes we consider the sweep-line method to be the most appropriate reduction method. The reason is that the bit-state hashing method does not guarantee full coverage of the state space which means that it cannot (in general) be used to obtain the full protocol language. The state-space caching method does explore the full state space which means

that the protocol language could be obtained. However, the state space caching method may visit a state several times which means that the FSA to be input into the FSM tool would be unnecessarily large.

The remainder of this paper is organised as follows: Section 2 provides background information on WAP, while Sect. 3 introduces the protocol design concepts central to the verification task. The Transaction Service and Protocol are described in Sect. 4 and 5, respectively. The full state space analysis of the protocol is presented in Sect. 6, along with the desired properties. Sections 7 to 10 present the application of the sweep-line method to the verification of WTP. Section 11 concludes with a summary and areas of future work.

2 Wireless Application Protocol

The Wireless Application Protocol (WAP) [21] is an architecture proposed by the WAP Forum [21] to overcome the difficulties of providing Internet services in wireless networks. The existing Internet protocols have been designed as a method for communicating across different physical, wired networks. However, several design assumptions no longer hold for wireless networks because of two main differences: wireless networks typically have higher error rates, lower bandwidth, and more frequent disconnections than their wired counterparts; and the devices typically used in wireless, mobile networks (e.g. mobile phones, PDAs) have smaller displays, shorter battery life, less powerful CPUs and alternative input devices than devices commonly used in wired networks (e.g. PCs). The objectives of the WAP architecture design are to take advantage of the existing knowledge and infrastructure in developing Web applications, while using protocols optimised to run over wireless links.

WAP is designed in layers as shown (shaded grey) in Fig. 1. There is an application layer at the top and then four protocol layers. The *Application layer* includes specification of a mark-up language suitable for displaying information on small screens, an accompanying scripting language, and a framework

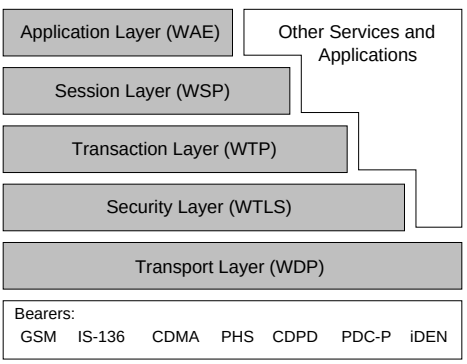


Fig. 1. WAP Architecture

for making use of telephony features in the device and network infrastructure. Together, the components form the Wireless Application Environment (WAE). The *Session layer* defines the Wireless Session Protocol (WSP) for creating a connection-less or connection-oriented session between a client and server. A connection-oriented session includes setup and tear-down, data push, and session suspend and resume. The *Transaction layer* defines the Wireless Transaction Protocol (WTP) which has three classes of service: an unreliable one-way request from the Initiator to the Responder (Class 0); a reliable one-way request (Class 1); and a reliable request by the Initiator and a reliable response from the Responder (Class 2). The *Security layer* defines the Wireless Transport Layer Security (WTLS) which provides applications with privacy, data integrity and authentication. This is an optional layer in the architecture. The *Transport layer* defines the Wireless Datagram Protocol (WDP) which provides a datagram service. WDP includes mappings to a range of supported bearer services so that the upper layer can operate independently of these bearers. The bearers supported include: GSM Short Message Service (SMS) and General Packet Radio Service (GPRS), CDMA Circuit Switched Data, Cellular Digital Packet Data (CDPD) and several proprietary protocols.

3 Protocol Design Concepts

Protocol engineering encompasses the design of communication architectures and protocols using formal methods [4]. We apply a protocol engineering methodology [3] to the analysis of WTP, so that a high level of confidence in the correctness of the design can be achieved. The methodology is based on a fundamental property of communication architectures: functionality is separated into a layered (or hierarchical) structure. This is illustrated in the WAP architecture of Fig. 1, where six layers are evident. The Open Systems Interconnection (OSI) [13] is an example of separation of functionality into layers. Most of the concepts used in our analysis of WTP are based on OSI [13]. Layering is achieved by describing two parts: the *service* the layer provides to the upper layer, and the *protocol* used within the layer to provide that service.

The service is described using *primitives* which convey information between service users. Service users submit primitives to the service provider and the provider delivers primitives to the users. The information conveyed by the primitive is given by its name, type and parameters. The four types of primitives are: request; indication; response and confirm. A complete service definition should specify all possible sets of primitive sequences that can occur between users. This is referred to as the *service language*.

The protocol in a layer describes the mechanisms for communicating between two (or more) protocol entities (PEs). The PEs communicate by sending *protocol data units* (PDUs) via the service provider of the layer below. The protocol describes: the actions each PE takes upon submission of a primitive by a service user; events that cause primitives to be delivered to users; and local events within each PE to ensure the service is provided.

To take advantage of the separation of functionality into layers, the protocol specification must provide the service described in the service specification. Proving this is true is known as verification and is the major focus of our analysis of WTP. Using CPNs, the state spaces of the service CPN and protocol CPN represent the possible sequences of events that can occur in the service and protocol, respectively. By treating both state spaces as FSA, and removing all events that do not correspond to primitives occurring, we can compare the two corresponding languages to determine if the protocol preserves the sequences of primitives defined in the service.

4 Transaction Service Specification

Our modelling and verification of WTP focuses on the Class 2 service provided by the Transaction layer which is: a reliable request from the *Initiator* and a reliable response from the *Responder*. The Class 2 Transaction Service defines the possible service primitives that can be submitted by or delivered to a user. There are three types of service primitives:

1. TR-Invoke: Initiates a new transaction. The types request (req), indication (ind), response (res) and confirm (cnf) are allowed.
2. TR-Result: Sends back a result of a previously initiated transaction. The req, ind, res and cnf types are allowed.
3. TR-Abort: Aborts an existing transaction. Only req and ind types are allowed.

The WTP Specification [22] describes the parameters for these primitives and gives a table that indicates when one primitive can follow another primitive. We explain the general concepts of a transaction using two sequences of primitives.

Figure 2(a) is a time sequence diagram that shows a possible primitive sequence representing a successful transaction. On the left the primitives submitted by and delivered to the Initiator user are shown. Similarly, the primitives seen by the Responder user are shown on the right. Dependencies between primitives are shown by dashed lines in the area representing the Transaction Service provider.

The Initiator user starts a transaction by submitting a TR-Invoke.req primitive. This is the only primitive that starts a transaction. A TR-Invoke.ind primitive is delivered to the Responder user, indicating that a request has been made. The Responder user then chooses to acknowledge the request, by submitting a TR-Invoke.res primitive. The acknowledgment, via the TR-Invoke.cnf primitive, indicates to the Initiator user that the Responder user has received the request.

Once the Responder user processes the request, (once completed) the user sends the result via the TR-Result.req primitive. The result is delivered to the Initiator user (TR-Result.ind), which then acknowledges its receipt (TR-Result.res). Finally receipt of delivery is confirmed (TR-Result.cnf) to the Responder user. The transaction has now been successfully completed.

There are several other sequences that represent a successful transaction. One scenario occurs when the Responder user does not explicitly acknowledge the invocation request (see Fig. 2(b)). Instead, the sending of the TR-Result.req

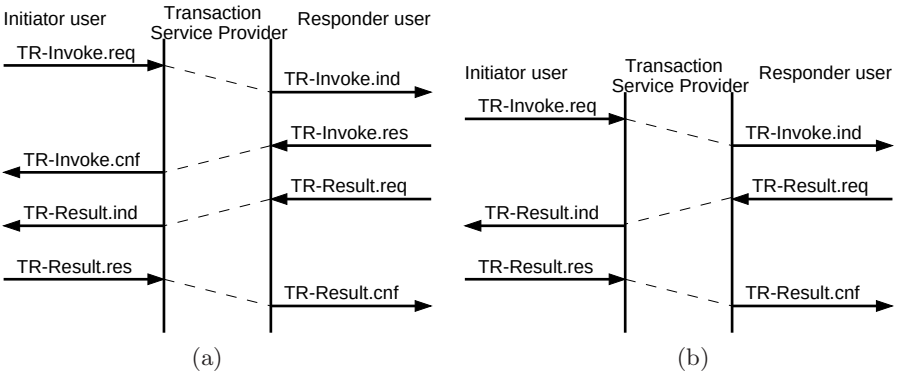


Fig. 2. Example service primitive sequences for UserAck On (a) and UserAck Off (b)

primitive, implicitly acknowledges that the invocation has been received. This is a more efficient scenario in terms of messages than that in Fig. 2(a). However, the sequence in Fig. 2(a) is allowed because it is required for use by the Wireless Session Protocol [21]. When it is required, the Initiator user sets a primitive parameter, UserAck, to On. With UserAck On, the users must acknowledge with the response primitives (TR-Invoke.res and TR-Result.res). When UserAck is Off, the response primitives may or may not be submitted.

We have presented two example primitive sequences illustrating successful transactions. There is also the possibility of transactions being aborted by either user or the Transaction Service provider. For further details on the Transaction Service, we refer the reader to [22,9,11].

The Transaction Service presented in the WTP Specification only specifies the possible primitive sequences from each user's point of view. There is no information describing how the two users coordinate with each other (although for some primitive sequences it is obvious). In both [9] and [11] we have discussed several shortcomings of the WTP Specification, which includes the absence of information regarding the global sequences of primitives. A CPN model of the Transaction Service has been created [11] with the intent of providing an unambiguous specification of the primitive sequences, and also to allow the automatic generation of the global set of sequences. For the full presentation of this model the reader is referred to [11]. The end result is the generation of two Transaction Service languages: one with UserAck Off and the other with UserAck On.

When UserAck is Off, there are 182 valid sequences of service primitives, the shortest sequence consists of two primitives and the longest consists of eight primitives. Figure 3 shows the FSA representing this language. The FSA was obtained by converting the Design/CPN state space representation into a text format used by the FSM tool [1], hide (make invisible) the non-primitive events, and minimise the FSA. The primitives are abbreviated using the first letter of the primitive name (I for Invoke, R for Result, and A for Abort) and the primitive

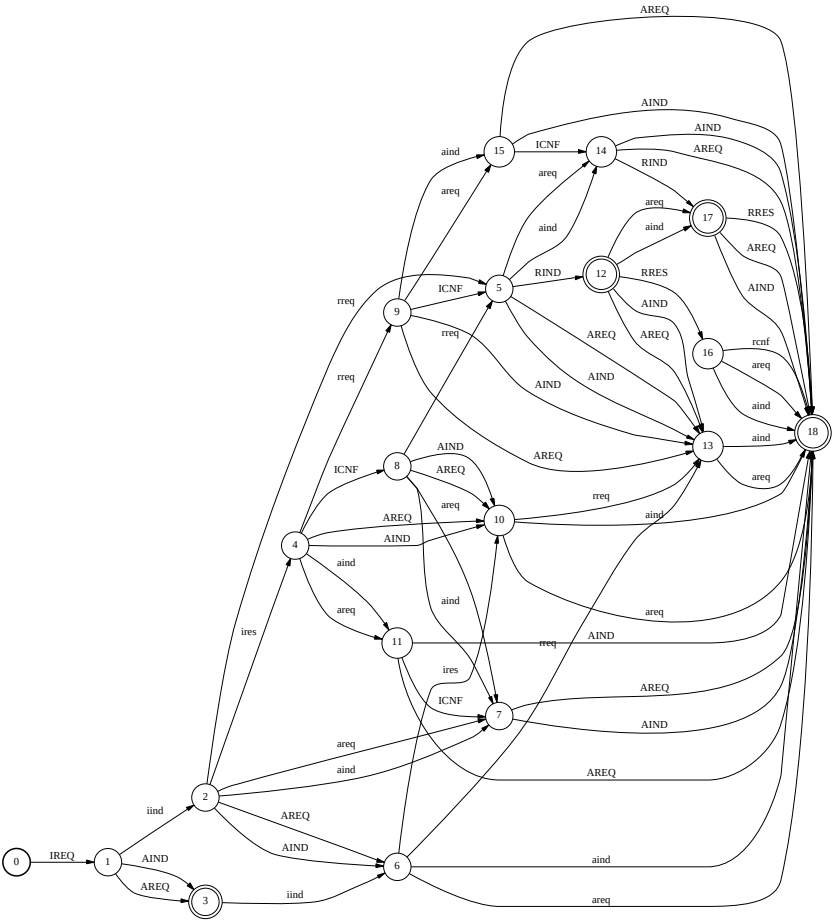


Fig. 3. Transaction Service language with UserAck Off

type (req, ind, res, cnf). Primitives seen by the Initiator user are in upper case, while the Responder user primitives are in lowercase. The initial state of the FSA is represented by node 0. Acceptance (or halt) states are indicated using double circles. When UserAck is On, the service language is a subset (with only 130 sequences) of the UserAck Off language. The reason for this is that the use of the response primitive is optional with UserAck Off. The two languages are used as the baseline for determining if the protocol refines the Transaction Service.

5 Revised Transaction Protocol Specification

The Transaction Protocol describes the operation of the two protocol entities, Initiator and Responder, in terms of PDUs that are used and the actions to be taken when events occur. In Sect. 5.1 we give a brief overview of the important

features of the protocol [22], especially those that impact the analysis results. Section 5.2 summarises the errors and the changes that lead to the Revised Transaction Protocol, which is analysed using full state spaces in Sect. 6 and using the sweep-line method in Sect. 7. Parts of the CPN model of the Revised Transaction Protocol are presented in Sect. 5.3.

5.1 Overview of Protocol

The Transaction Protocol defines the procedures for the Initiator PE and Responder PE to communicate in order to provide the Transaction Service. There are four types of PDUs sent between the PEs:

Invoke: Sent by the Initiator PE to start a transaction.
Result: Sent by the Responder PE to return the result.
Ack: Sent by either PE to acknowledge the Invoke PDU or Result PDU.
Abort: Sent by either PE to abort the transaction.

The PDUs are sent via the Transport Service Provider which in the WAP architecture is a datagram protocol. It is assumed re-ordering, losses and duplication of PDUs can occur in the Transport Service Provider. Each PDU has a set of parameters. The important parameters (in terms of the analysis results), along with the general procedure for a successful transaction, will become apparent from the following example (see Fig. 4).

Figure 4 is a time sequence diagram that shows: the primitives submitted by and delivered to the users; events that take place at the PEs (which are represented by the two vertical lines); and the PDUs that are transmitted via the Transport Service Provider. Time increases from top to bottom. The first event shown is the Initiator user submitting a TR-Invoke.req primitive to start a transaction. We assume the UserAck parameter is set to On.

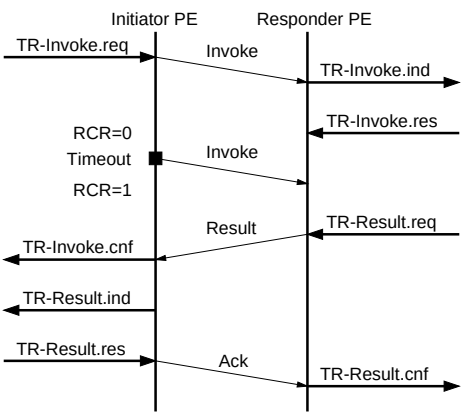


Fig. 4. Example sequence of protocol events for successful transaction

Upon submission of the TR-Invoke.req primitive, the Initiator PE sends an Invoke PDU via the Transport Service Provider. A common parameter to every PDU is a *transaction identifier* (TID). The TID of PDUs in one transaction are identical. When the Initiator user initiates a new transaction (which can occur before the previous transaction has been completed) a new TID is given to the PDUs. The WTP Specification describes how the TID parameter is used to ensure PEs do not confuse which transaction the PDUs belong to.

Upon receipt of the Invoke PDU, the Responder PE delivers a TR-Invoke.ind primitive to the Responder user. The Responder user then submits a TR-Invoke.res primitive, acknowledging the invocation. Note that the Responder PE does not immediately send an acknowledgment to the Initiator PE (to reduce the number of PDUs sent across the wireless link). Instead the Responder PE waits for the TR-Result.req from the Responder user. However, in the meantime, as the Initiator PE has not received an acknowledgment of the invocation, it assumes the Invoke PDU was lost, and therefore re-transmits. The re-transmission occurs after a timer at the Initiator PE expires. The Initiator PE also increments a counter, called the *Retransmission Counter* (RCR), which limits the PE to $RCRI_{\max}$ retransmissions, after which the transaction will be aborted.

As the Responder PE has already received an Invoke PDU, it discards the re-transmitted Invoke PDU. When the Responder user submits the TR-Result.req primitive, a Result PDU is sent to the Initiator PE. Upon its receipt, the TR-Invoke.cnf and TR-Result.ind primitives are delivered to the Initiator user (i.e. the user now has the result). The last step in the transaction is the acknowledgment of the result. The Initiator user submits the TR-Result.res primitive, the Initiator PE sends an Ack PDU and upon its receipt a TR-Result.cnf primitive is delivered to the Responder user. The transaction has now been successfully completed. Note that the Responder PE also maintains a retransmission counter which it uses to re-transmit the Result PDU if it has not received an acknowledgment before a certain timeout period. The number of retransmissions in the Responder is limited to a number denoted by $RCRR_{\max}$.

The example in Fig. 4 has shown the general procedure for a transaction. There are many other scenarios, due to other features being used such as: multiple transactions can be underway at the same time; transactions can be aborted either by the users or the service provider; PDUs can be received out of order, or not received at all; and the UserAck option is set to Off.

The procedures for the Transaction Protocol's operation are mainly described in [22] using a set of state tables. There is a table for each state of each PE, and each entry (row) in the table comprises four items (columns): the event, such as the submission of a primitive by a user or timeout at the PE; conditions on the event; actions to be taken; and the next state of the PE. These state tables are used as the basis of the CPN model of the Transaction Protocol.

5.2 Errors in the Transaction Protocol

The approach to analysing WTP is to model the protocol described in the WTP Specification [22], compare the language to the Transaction Service language

(and prove other properties, which are discussed in Sect. 6), and incrementally fix any errors present. In [10] three errors were presented, and fixes proposed. Since then a number of other errors have been discovered in the Transaction Protocol [11]. We summarize the two main errors here.

1. The Ack PDU and Result PDU are ambiguous, in that the receiving PE cannot be certain if the PDU indicates that the peer user has acknowledged (submitted a response primitive), or only the peer PE has acknowledged (i.e. the response primitive has not been submitted). Both PDUs require an extra field to explicitly indicate their meaning.
2. After a transaction has been completed, the Responder PE may receive a re-transmitted Invoke PDU or Ack PDU after certain messages have been lost, and wrongly accept it to be part of a new transaction. The mechanism for checking the validity of Invoke PDUs must be prevented from taking place until a set time after a transaction has completed (instead, the Invoke PDUs are discarded). After that time, the Responder PE will know that any Invoke PDUs will be for new transactions.

These two errors do not change the general procedure for a transaction as described in Sect. 5.1. Further details on the errors and proposed solutions can be found in [11]. This motivated the development of a revised Transaction Protocol to eliminate these errors. To see if the revised protocol was correct, we revised the previous CPN model accordingly.

5.3 Revised Protocol Specification CPN

The CPN of the Revised Transaction Protocol comprises four hierarchical levels (see Fig. 5), starting with a single page (TR_Protocol) giving an overview of the two PEs and the communication channel. The communication channel is modelled as a place for either direction of communication: InitToResp and RespToInit. To limit the size of the state space, a bound has been put on each of the two places InitToResp and RespToInit for the number of PDUs that can be in transit between the Initiator and the Responder. When the bound is larger than 1, reordering of PDUs in the Transport Service Provider is possible. Moreover, PDUs may be lost in transmission. One limitation of the model is that we currently do not consider duplication of PDUs. Another is the omission of the segmentation and re-assembly feature (which is optional in the protocol). The CPN is also simplified by considering just a single transaction, instead of multiple, concurrent transactions. This is justified because using TIDs for PDUs the transactions are independent, provided that PDUs have a maximum packet lifetime (which is assumed in the specification).

There are two second level pages, TR_Init_PE and TR_Resp_PE, which show the major states of each PE. Figure 6 shows the Initiator PE page. The two communication places are shown on the right, substitution transitions represent the procedures that occur in the set of major states, and the place Initiator maintains state information for the PE. The initial marking of Initiator

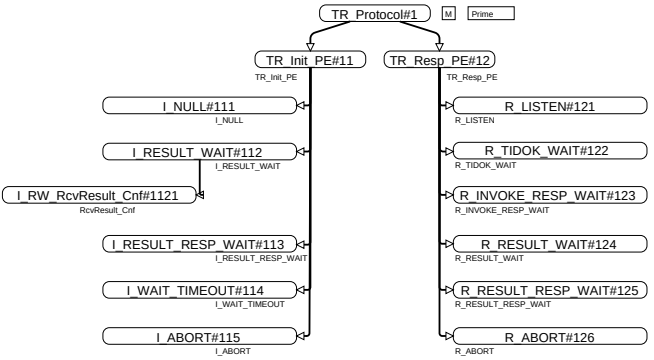


Fig. 5. Revised Transaction Protocol CPN hierarchy page

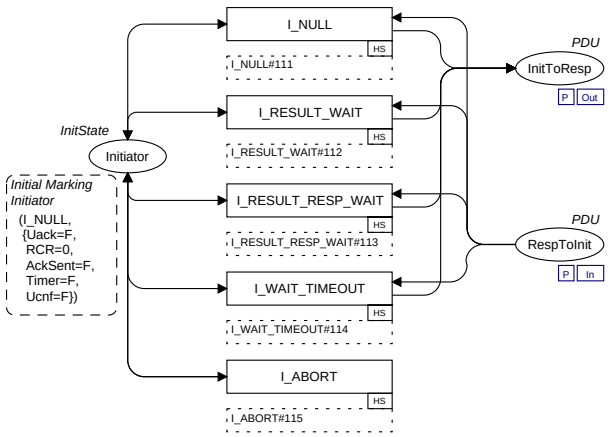


Fig. 6. Revised Transaction Protocol CPN Initiator PE page

shows the Initiator PE is in the I_NULL state and gives the initial values of the state variables (e.g. RCR=0). The general procedure for the Initiator PE during a single transaction is to traverse through the states in the following order: I_NULL, I_RESULT_WAIT, I_RESULT_RESP_WAIT, and I_WAIT_TIMEOUT. When the transaction is completed, the Initiator PE returns to the I_NULL state with certain of its other state variables set to special values. The state I_ABORT models a Transport Service Provider initiated abort. The Responder PE has a similar page, where its major states are represented by substitution transitions.

Each transition on the TR_Init_PE page (Fig. 6) is decomposed into another CPN page. These third level pages model the entries in the state tables given in the WTP Specification. Figure 7 shows the I_NULL page for the Initiator PE. The general approach is to have a transition for every entry in the state tables given in [22]. Each transition has an input arc from the Initiator place to ensure it can only be enabled when the Initiator PE is in the correct state and the

state variables meet the conditions of the state table entry. Transitions model primitives being submitted by the user, primitives being delivered to the user (the only fourth level page models a special case of this), timeouts, the receipt of PDUs and the sending of PDUs. The full Transaction Protocol CPN (original and revised) can be found in [11] for the case when the medium does not lose or duplicate PDUs.

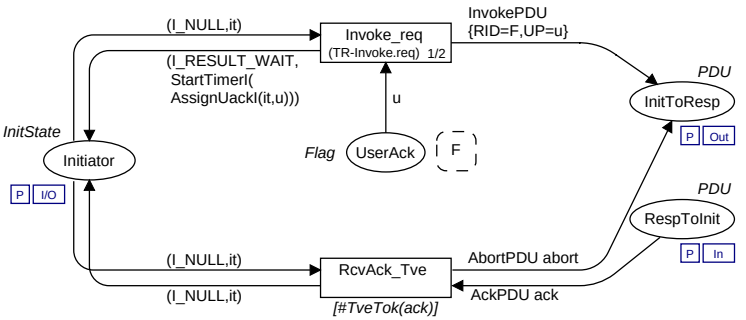


Fig. 7. Revised Transaction Protocol CPN I_NULL page

6 Full State Space Analysis

The verification of the Revised Transaction Protocol using full state spaces involves proving four desired properties (presented in Sect. 6.1), then obtaining the protocol primitive language from the state space and comparing it to the Transaction Service language. As there are several input parameters for the protocol this process is repeated for a set of parameter values. The state space and language analysis results are summarised in Sect. 6.2.

6.1 Desired Properties of the Revised Transaction Protocol

We have defined [11] four desired properties of the Revised Transaction Protocol that should hold to give a high level of confidence in the correctness of the design. The first, and most important, is the faithful refinement of the Transaction Service. The state space of the protocol is computed and converted to a textual representation using Design/CPN [7], which is then used as input to FSM [1] to obtain the protocol language. The Transaction Protocol language and the Transaction Service language are then compared using FSM, and they must be identical for this property to be true. FSM implements standard algorithms for determinisation and comparison of minimal FSAs to check language equivalence.

The three other properties are: successful termination; absence of livelocks and absence of unexpected dead transitions. We have specified the expected dead markings (called terminal markings) as a predicate on the markings, and

for successful termination the state space must only contain these terminal markings. No deadlocks (i.e. undesired dead markings) can be present. The strongly connected components (SCC) graph of the Transaction Protocol is used to determine the absence of livelocks. We define livelocks as terminal strongly connected components of the SCC graph with more than one node or one or more arcs. For some sets of parameters several features of the protocol may not be activated. As a result, transitions modelling these features will be dead. We have identified such transitions, and for the fourth property to be true, no other transitions can be dead.

6.2 Analysis Results

There are four parameters used in the Revised Transaction Protocol CPN: $RCRI_{\max}$, the maximum value of the counter RCR used by the Initiator PE; $RCRR_{\max}$, the maximum value of the RCR used by the Responder PE; and *UserAck*, which indicates whether the transaction has UserAck On or Off. Finally, there is the bound on the places modelling the communication channel. We will denote this bound by C_{chan}. Ideally, the properties would be verified independently of the parameters. However, our goal is to investigate configurations that are likely to be used in practice. The WTP Specification gives several suggested values for the counter parameters: $RCRI_{\max}$ and $RCRR_{\max}$ both 8 for IP (Internet Protocol) networks; and 4 in GSM SMS networks [22]. Here we will consider the configuration for which $RCRI_{\max} = RCRR_{\max}$ assuming that the Initiator and Responder work across the same network. All results presented in this paper were obtained on a PIII 1GHz PC with 512 Mb of memory.

Table 1 shows the experimental results obtained for values of the retransmission counters up to 5 which include those recommended for GSM SMS networks. The Config column gives the values of the parameters considered written in the form $X - Y$ where X specifies $RCRI_{\max}$, and Y specifies whether UserAck is On (indicated by T) or Off (indicated by F). The bound on the communication channel C_{chan} was in all cases set to 2 to consider re-ordering of PDUs. The Nodes and Arcs columns give the number of nodes and arcs in the full state space, respectively. The G-Time column gives the time for generation of the state space and for checking the properties of absence of deadlocks, absence of livelocks, and absence of unexpected dead transitions. The generation time is written in the form h:mm:ss where h is hours, mm is minutes and ss is seconds. The time used for verification of the three properties was in all cases insignificant compared to the state space generation time.

The C-Time column gives the time for checking (using FSM) whether the protocol and the service languages are identical. It is worth observing that the time used for language comparison was in all cases significantly less than the time used for state space generation in Design/CPN. In all cases, the deterministic FSA computed from the state space was smaller than the full state space. This is the main reason why comparison of the languages is tractable in this case.

For all configurations in Table 1, the protocol and service languages were equivalent, and the protocol successfully terminated and did not include livelocks

or unexpected dead transitions. The results are only given for $RCRI_{\max}$ and $RCRR_{\max}$ up to 5 because Config 6-F could not be calculated using full state space analysis on the machine used. Configs 6-T and 7-T could be calculated, and the properties were proven correct. Therefore, five is the highest value of the counters for which the protocol can be verified with both UserAck On and Off.

7 Sweep-Line Analysis

There is an intuitive presence of progress in the Transaction Protocol from the start of the transaction towards the transaction either being successfully completed or aborted. This progress is (for instance) explicit in the Initiator PE that starts in its initial state (I_NULL) and progresses through its intermediate internal states (I_RESULT_WAIT, I_RESULT_RESP_WAIT, I_WAIT_TIMEOUT) (see Fig. 6) and eventually ends in the I_NULL state with its state variables set to special values. We will denote the terminating state of the Initiator by I_TERM. There is a similar presence of progress in the Responder PE. In addition to the progress in the internal states of the two protocol entities, there is also progress due to the retransmission counters in the two protocol entities as the values of these increases as the transaction proceeds.

The progress exhibited by the Transaction Protocol is also reflected in the state space of the CPN model. Figure 8 shows the initial fragment of the state space. To simplify the figure we have omitted states in this initial fragment resulting from user and provider initiated abort. The states have been organised into four layers based on how far the system has *progressed* according to the internal state of the two protocol entities. For example, layer 1 (the top most layer) contains the states where the Initiator PE is in state I_NULL and the Responder PE is in state R_LISTEN. The initial marking represented by node 1 is the only marking in this layer. Nodes 2-4 constituting layer 2 are the states where the Initiator PE is in state I_RESULT_WAIT and the Responder PE is in R_LISTEN. Hence, the lower the layer, the further the system has progressed.

The key observation to make is that progress in the Transaction Protocol manifests itself by the property that a marking in a given layer has successor markings either in the same layer or in some lower layer, but never in an upper layer. The idea underlying the sweep-line method [6] is to exploit such progress

Table 1. Experimental results for full state space analysis

Config	Nodes	Arcs	G-Time	C-Time	Config	Nodes	Arcs	G-Time	C-Time
1-T	1,838	9,587	0:00:03	0:00:01	1-F	4,232	32,686	0:00:10	0:00:01
2-T	10,333	58,286	0:00:30	0:00:02	2-F	24,905	149,792	0:01:52	0:00:03
3-T	30,978	178,329	0:02:15	0:00:08	3-F	74,017	453,010	0:10:43	0:00:39
4-T	65,873	380,825	0:07:06	0:00:21	4-F	154,231	945,127	0:38:15	0:00:46
5-T	173,343	654,842	0:16:33	0:00:47	5-F	262,442	1,605,984	1:36:38	0:01:43

by deleting markings on-the-fly during state space exploration. The deletion is done such that the state space exploration will eventually terminate and upon termination all reachable markings will have been explored exactly once.

To illustrate how the sweep-line method operates, consider Fig. 8 and assume that it represents a snapshot taken during conventional state space exploration. Dashed nodes are *fully processed* markings (i.e. markings that are stored in memory and all their successor markings have been calculated). Nodes with a thick solid black border are *unprocessed* nodes (i.e. nodes that are stored in memory, but their successor markings have not yet been calculated). Nodes with a thin solid black border are markings that have not yet been calculated.

If the state space exploration algorithm processes markings according to the progress of the protocol they correspond to, node 4 will be the marking among the unprocessed markings that will be selected for processing next. This will add nodes 14, 15, 20, and 21 to the set of stored markings and mark these as unprocessed. At this point it can be observed that it is not possible from any of the unprocessed markings to reach one of the markings 1, 2, 3, and 4. The reason is that nodes 1, 2, 3, and 4, represent markings where the protocol has not progressed as far as in any of the unprocessed markings. Hence, it is safe to delete nodes 1, 2, 3, and 4, as they cannot possibly be needed for comparison with newly generated markings when checking (during the state space exploration) whether a marking has already been visited. In a similar way, once all the markings in the third layer have been fully processed these nodes can be deleted from the set of nodes stored in memory. Intuitively, one can think of a *sweep-line* being aligned with the highest layer (seen from the top) that contains unprocessed markings. During state space exploration, unprocessed markings are selected for processing in a least-progress first order causing the sweep-line to move downwards. Markings will thereby be added in front of the sweep-line and deleted behind the sweep-line.

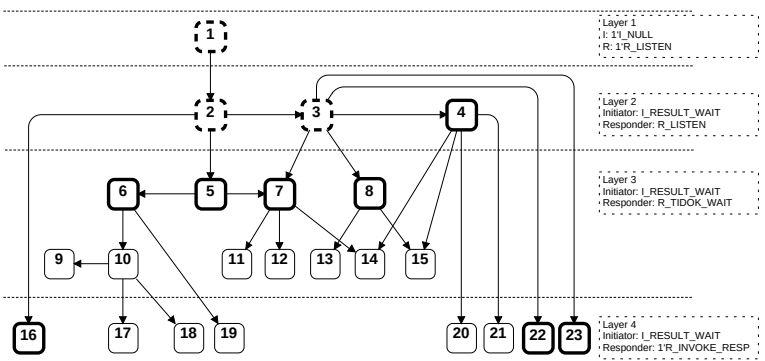


Fig. 8. Initial fragment of the state space for the Revised Transaction Protocol

8 Progress Measure Specification

The progress exploited by the sweep-line method is formally captured by a *progress measure*. In this section we formally specify the progress measure for the Revised Transaction Protocol based on the intuition presented in Sect. 7.

A *progress measure* [6] consists of a *progress mapping* assigning a *progress value* to each marking, and a *partial order* $(O, \sqsubseteq)$ on the markings of the system. A partial order $(O, \sqsubseteq)$ consists of a set O and a relation $\sqsubseteq \subseteq O \times O$ which is reflexive, transitive, and antisymmetric. Moreover, the partial order is required to preserve the reachability relation of the system. The definition of progress measure below is identical to Def. 1 in [6] except that we give the definition in a Petri Net formulation. In the definition, M_0 denotes the initial marking, $[M_0\rangle$ denotes the set of markings reachable from the initial marking, and $\mathbb{M}$ denotes the set of all markings. If a marking M_1 is reachable from a marking M_2 via some occurrence sequence we write $M_1 \rightarrow^* M_2$. In particular $M \rightarrow^* M$ for all markings $M \in \mathbb{M}$.

Definition 1. *A* ***progress measure*** *is a tuple* $\mathcal{P} = (O, \sqsubseteq, \psi)$ *such that* $(O, \sqsubseteq)$ *is a partial order and* $\psi : \mathbb{M} \rightarrow O$ *is a progress mapping from markings into* O *satisfying:* $\forall M, M' \in [M_0\rangle : M \rightarrow^* M' \Rightarrow \psi(M) \sqsubseteq \psi(M')$. □

It is worth noting that the definition of progress measure implicitly states that for all $M \in [M_0\rangle : \psi(M_0) \sqsubseteq \psi(M)$, i.e., the initial marking is minimal among the reachable markings with respect to the progress measure. The requirement that the progress measure preserves the reachability relation can be verified fully automatically during the sweep-line state space exploration since each arc of the state space is explored. From Def. 1 it follows that the sweep-line method can handle state space containing cycles, but all states on the cycle are required to have the same progress measure.

For specifying Transaction Protocol progress measure, we introduce some notation: For a reachable marking M, $\mathrm{RCRI}(M)$ denotes the value of the Initiator PEs retransmission counter in M, $\mathrm{RCRR}(M)$ denotes the value of the Responder PEs retransmission counter in M, $\mathrm{Is}(M)$ denotes the Initiator PE's internal state in M, and $\mathrm{Rs}(M)$ denotes the Responder PE's internal state in M. To capture the progress from the internal states, we define a mapping ψ_{Is} that enumerates the Initiator PE's states according to the progress it represents.

$$\psi_{\mathrm{Is}}(M) = \begin{cases} 0 \text{ if } \mathrm{Is}(M) = \mathsf{I_NULL} \\ 1 \text{ if } \mathrm{Is}(M) = \mathsf{I_RESULT_WAIT} \\ 2 \text{ if } \mathrm{Is}(M) = \mathsf{I_RESULT_RESP_WAIT} \\ 3 \text{ if } \mathrm{Is}(M) = \mathsf{I_WAIT_TIMEOUT} \\ 4 \text{ if } \mathrm{Is}(M) = \mathsf{I_TERM} \end{cases}$$

A similar mapping (denoted ψ_{Rs}) is defined enumerating the Responder PE's internal state according to progress. We have not assigned any value to I_ABORT (see Fig. 6) since it only represents the event of an abort and not an actual state of the Initiator PE.

We define two mappings ψ_{RCRI} and ψ_{RCRR} to capture progress in terms of the retransmission counters in the Initiator PE and the Responder PE, respectively. The definitions are: $\psi_{\text{RCRI}}(M) = \text{RCRI}(M)$ and $\psi_{\text{RCRR}}(M) = \text{RCRR}(M)$.

In addition to the progress due to retransmission counters and internal states of the protocol entities, we will also exploit that when both protocol entities have terminated, the sum of the number of tokens on the two places InitToResp and RespToInit is decreasing (see Fig. 6). The mapping ψ_{CHAN}, capturing progress in terms of the decreasing number of tokens on the communication channel, is defined below. For a place p and a marking M, we denote by $M(p)$ the marking of p in M, and by $|M(p)|$ the number of tokens on p in M.

$$\psi_{\text{CHAN}}(M) = 2 * C_{\mathsf{chan}} - (|M(\mathsf{InitToResp})| + |M(\mathsf{RespToInit})|)$$

Having identified the sources of progress, we can now define the full progress mapping ψ_{WTP} for the Transaction Protocol as a 5-dimensional vector:

$$\psi_{\mathsf{WTP}}(M) = \begin{cases} (\psi_{\text{Is}}(M), \psi_{\text{Rs}}(M), \psi_{\text{RCRI}}(M), \psi_{\text{RCRR}}(M), \psi_{\text{CHAN}}(M)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \text{Is}(M) = \mathsf{I_TERM} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \text{Rs}(M) = \mathsf{R_TERM} \\ (\psi_{\text{Is}}(M), \psi_{\text{Rs}}(M), \psi_{\text{RCRI}}(M), \psi_{\text{RCRR}}(M), 0) \qquad\quad \text{otherwise} \end{cases}$$

For the definition of the partial order $\sqsubseteq_{\mathsf{WTP}}$ between the progress values, we view the progress values as *progress vectors* in $\mathbb{Z}^5$ and consider one progress vector $\psi_{\mathsf{WTP}}(M_1) = (a_1, a_2, a_3, a_4, a_5)$ to be less than or equal to a progress vector $\psi_{\mathsf{WTP}}(M_2) = (b_1, b_2, b_3, b_4, b_5)$ if and only if it is possible to obtain $\psi_{\mathsf{WTP}}(M_2)$ from $\psi_{\mathsf{WTP}}(M_1)$ by increasing the progress values (coordinates) in $\psi_{\mathsf{WTP}}(M_1)$. Formally the ordering is defined as: $\psi_{\mathsf{WTP}}(M_1) \sqsubseteq_{\mathsf{WTP}} \psi_{\mathsf{WTP}}(M_2)$ if and only if $a_i \leq b_i$ for all $1 \leq i \leq 5$. It is straightforward to see that $\sqsubseteq_{\mathsf{WTP}}$ is a partial order. That the tuple $(\psi_{\mathsf{WTP}}, (\sqsubseteq_{\mathsf{WTP}}, \mathbb{Z}^5))$ is indeed a progress measure, i.e., also preserves the reachability relation according to Def. 1, is checked fully automatically by the Design/CPN sweep-line library [6] that we apply for the state space exploration.

9 Implementing the Progress Measure

In this section we show how the progress measure ψ_{WTP} in the previous section can be implemented and provided as input to the sweep-line library [6]. The sweep-line library is implemented on top of Design/CPN [7], and the STANDARD ML (SML) [18] programming language is available to the user for specifying progress measures. The user provides a progress measure to the tool by writing an SML function mapping a marking into an integer. SML support for unbounded integers (integers which cannot overflow) is exploited as the co-domain for progress mappings. The ordering on progress values is the usual total ordering on integers.

The first step when implementing ψ_{WTP} is to overcome the limitation that the sweep-line library currently only supports progress measures where the progress

values are integers and the ordering is the total ordering on integers. The progress values for ψ_{WTP} are vectors, and the ordering $\sqsubseteq_{\mathsf{WTP}}$ is a partial order. Hence we need to define a (preferably injective) mapping from $\mathbb{Z}^5$ to $\mathbb{Z}$ and embed the partial order $\sqsubseteq_{\mathsf{WTP}}$ into the total ordering ($\leq$) on integers. A simple way to do this is to interpret the vector $\psi_{\mathsf{WTP}}(M) = (a_1, a_2, a_3, a_4, a_5)$ as a number in base n, where n is larger than any value that can be assumed by the components. Hence, we can set $n = \max(2{*}C_{\mathsf{chan}}, 4, RCRI_{\max}, RCRR_{\max}){+}1$. The embedded progress measure $\hat{\psi}_{\mathsf{WTP}}$ can be defined as:

$$\hat{\psi}_{\mathsf{WTP}}(M) = \psi_{\mathrm{Is}}(M) * n^0 + \psi_{\mathrm{Rs}}(M) * n^1 + \psi_{\mathrm{RCRI}}(M) * n^2 + \psi_{\mathrm{RCRR}}(M) * n^3 + \psi_{\mathrm{CHAN}}(M) * n^4$$

Another possibility would be to define the embedded progress values as the Euclidean distance from $(0, 0, 0, 0, 0)$ to the point $\psi_{\mathsf{WTP}}(M)$. Also, the weight given to the different components can be set differently. In the rest of this paper we used the embedding specified above. It is left as future work to compare the alternatives. The embedded progress measure can be implemented in 20 lines of SML code and input into the sweep-line library. It should be noted that by embedding the progress measure into the set of integers (i.e. making it 1-dimensional) we "lose" some of the progress in the system compared to being able to handle the 5-dimensional progress measure directly.

10 Sweep-Line Experimental Results

The sweep-line method supports on-the-fly verification of safety and reachability properties. Checking for dead markings and identifying dead transitions is straightforward with the sweep-line method since all nodes and arcs of the state space are visited. The sweep-line library has direct support for checking that all dead markings satisfy a certain marking predicate, and for identifying the dead transitions. Hence, checking for absence of deadlocks and that only the desired transitions are dead is fully supported by the sweep-line library. The absence of livelocks can also be checked with the sweep-line method, but is currently not supported by the sweep-line library. Hence, for the configurations of WTP where verification has been done using the sweep-line method, we have not been able to check the absence of livelock property. To compare the protocol and the service languages, we write the FSA corresponding to the state space into a file *during* the state space exploration. This file can then be used as input to the FSM tool. This means that the full state space is represented in the FSM tool, but it still saves memory in practice since the information related to a node and an arc is reduced to an integer, whereas in the current implementation of the state space method in Design/CPN, a state takes up more space than an integer.

Table 2 shows the experimental results obtained with the sweep-line method for different configurations of WTP with values for $RCRI_{\max} = RCRR_{\max}$ up to 8, and $C_{\mathsf{chan}} = 2$. The Full State Space column lists (for comparison) the number of nodes in the full state space and the time for full state space generation

Table 2. Experimental results for sweep-line analysis

Config	Full State Spaces		Sweep-Line Method				
	Nodes	Time	Peak Nodes	GC-Time	E-Time	Threshold	C-Time
1-T	1,838	0:00:03	762 (41.4)	0:00:01	0:00:04	250	0:00:01
2-T	10,333	0:00:30	3,873 (37.5)	0:00:02	0:00:26	1500	0:00:01
3-T	30,978	0:02:15	9,982 (32.2)	0:00:10	0:01:29	3000	0:00:08
4-T	65,873	0:07:06	17,543 (26.6)	0:00:09	0:01:35	4500	0:00:20
5-T	113,343	0:16:33	27,737 (24.5)	0:00:59	0:06:04	6000	0:00:46
6-T	172,657	0:32:45	39,552 (22.9)	0:01:56	0:09:50	7500	0:01:29
7-T	243,765	1:07:52	53,985 (22.1)	0:02:56	0:14:21	10000	0:02:31
8-T	326,667	-	70,037 (21.4)	0:04:25	0:20:13	12500	0:06:00
1-F	4,232	0:00:10	1,857 (43.9)	0:00:01	0:00:11	500	0:00:01
2-F	24,905	0:01:52	9,099 (36.5)	0:00:08	0:01:17	3000	0:00:04
3-F	74,017	0:10:43	23,445 (31.7)	0:00:38	0:04:21	6000	0:00:16
4-F	154,231	0:38:15	41,145 (26.7)	0:01:48	0:09:54	9000	0:00:47
5-F	262,442	1:36:38	63,627 (24.2)	0:04:00	0:18:48	12000	0:01:46
6-F	397,583	-	89,970 (22.6)	0:07:15	0:29:52	15000	0:04:36
7-F	559,604	-	121,333 (21.7)	0:10:26	0:43:04	20000	0:16:16
8-F	748,505	-	160,091 (21.4)	0:14:40	0:59:09	25000	2:36:07

when possible. A "-" is an entry that indicates that the corresponding number could not be obtained. The Sweep-Line Method columns list the peak number of markings stored with the sweep-line method. The number in parentheses after the peak number of markings gives (in percentage) the peak number of markings divided by the number of markings in the state space. The GC-Time column gives the time used by the sweep-line method for garbage collection of states and the E-Time column gives the total time for the exploration of the state space, including the time spent in writing the FSA into a file. The Threshold column gives the garbage collection threshold used for the state space exploration. The sweep-line library uses a simple algorithm for initiating garbage collection during the sweep: whenever n new markings have been added to the state space, garbage collection is initiated. The C-Time column lists the time used by the FSM tool to compare the protocol and the service languages. It can be seen that the use of the sweep-line method reduces the peak number of nodes stored during state space exploration ranging from around 44 % to 21 %. Also, the total calculation time using the sweep-line method is significantly shorter than using the full state space analysis due to the fewer number of markings that a new marking is compared against. All comparisons revealed that the protocol and services languages were equivalent. It is also interesting to observe that as the configurations are approaching counter values of 8, the language comparison starts to become the bottleneck rather than the generation of the state space.

11 Conclusions

We have applied the state space method for Coloured Petri Nets and language comparison techniques as implemented in the FSM tool for the verification of WTP focusing on checking that the revised version of WTP conforms to the service specification. Full state spaces were used to verify WTP for smaller configurations and the sweep-line method was applied to obtain the state spaces for the larger configurations. We have now verified WTP for the two configurations recommended in the WTP Specification, giving us a high level of confidence in the correctness of the design. This illustrates the applicability of formal methods to identify and fix errors in communication protocols, and prove properties for non-trivial and practically relevant scenarios. The use of the sweep-line method made it possible to reduce peak memory consumption of Design/CPN to about 20 % of the full state space. This allowed us to obtain the state spaces for the configurations of interest. We have shown how control flow and retransmission counters in protocols constitute a source of progress that can be exploited to obtain a progress measure for the sweep-line method. This technique is applicable in general for verification of protocols using the sweep-line method.

A disadvantage of our current approach is that the comparison between the service and the protocol language is not done on-the-fly during the state space exploration. We have to write the FSA corresponding to the protocol language into a file such that it could be compared with the service language using FSM. Future work will investigate whether a method exists for the protocol language to be compared with the service language during the state space exploration with the sweep-line method.

References

1. AT&T. FSM Library. Web site: `http://www.research.att.com/sw/tools/fsm`. 183, 187, 193
2. J. Billington, M. Diaz, and G. Rozenberg (Eds.) *Application of Petri Nets to Communication Networks*. LNCS 1605. Springer-Verlag, 1999. 182
3. J. Billington, M. C. Wilbur-Ham, and M. Y. Bearman. In *Protocol Specification, Verification and Testing V*, pages 59–70. Elsevier Science Publishers, Amsterdam, New York, Oxford, 1986. 182, 185
4. J. Billington. Formal Specification of Protocols: Protocol Engineering. In *Encyclopedia of Microcomputers*, pages 299–314. Marcel Dekker, New York, NY, 1991. 185
5. S. Christensen, L. M. Kristensen, and T. Mailund. *Design/CPN Sweep-Line Method Library*. Department of Computer Science, Aarhus University, Aarhus, Denmark, 2001. To appear, `http://www.daimi.au.dk/designCPN/`. 183
6. S. Christensen, L. M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS 2001*, pages 450–464, LNCS 2031, Springer-Verlag, 2001 183, 195, 197, 198
7. CPN Group. Design/CPN Online. `http://www.daimi.au.dk/designCPN/`. 182, 193, 198

8. D. E. Comer. *Internetworking with TCP/IP*. Prentice Hall, Upper Saddle River, NJ, fourth edition, 2000.
9. S. Gordon and J. Billington. Modelling the WAP Transaction Service using Coloured Petri nets. In *Proc. of MDA 1999*, pages 105–114. LNCS 1748, Springer-Verlag, 1999. 183, 187
10. S. Gordon and J. Billington. Analysing the WAP Class 2 Wireless Transaction Protocol using Coloured Petri nets. In *Proc. of ICATPN 2000*, pages 207–226, LNCS 1825. Springer-Verlag, 2000. 183, 191
11. S. D. Gordon. *Verification of the WAP Transaction Layer using Coloured Petri Nets*. PhD thesis, University of South Australia, Australia, Nov. 2001. 183, 187, 191, 193
12. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991. 183
13. ITU. Information Technology—Open Systems Interconnection—Basic reference model: The basic model. ITU-T Recommendation X.200, July 1994. 185
14. C. Jard and T. Jeron. Bounded-memory Algorithms for Verification On-the-fly. In *Proc. of CAV'91*, pages 192–202, LNCS 575. Springer-Verlag, 1991. 183
15. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volumes 1 to 3*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997. 182, 183
16. J. B. Jørgensen and L. M. Kristensen. Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm using Coloured Petri nets and Occurrence Graphs with Symmetries. *IEEE Trans. Parallel and Dist. Sys.*, 10(7):714–732, July 1999. 183
17. L. Lorentsen and L. Kristensen. Modelling and analysis of a Danfoss Flowmeter System. In *Proc. of ICATPN'2000*, pages 346–366, LNCS 1825. Springer-Verlag, 2000. 183
18. J. D. Ullman. *Elements of ML Programming*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1998. 198
19. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets 1990*, pages 491–515, LNCS 424. Springer-Verlag, 1990. 183
20. A. Valmari. Stubborn Sets of Coloured Petri Nets. In G. Rozenberg, editor, *Proc. of ICATPN'91*, pages 102–121, 1991. 183
21. WAP Forum. Wireless Application Protocol.
Specifications available via: `http://www.wapforum.org/`. 184, 187
22. WAP Forum. WAP Wireless Transaction Protocol Specification. June 2000 Conformance Release. Available via: `http://www.wapforum.org/`, 19 Feb. 2000. 182, 183, 186, 187, 189, 190, 192, 194

Characterizing Liveness of Petri Nets in Terms of Siphons*

Li Jiao[1, 2], To-Yat Cheung[1], and Weiming Lu[2]

[1] Department of Computer Science, City University of Hong Kong
Hong Kong, China
jiao@cs.cityu.edu.hk
cscheung@cityu.edu.hk

[2] Academy of Mathematics and System Science, Chinese Academy of Sciences
Beijing, China
wmlu@math03.math.ac.cn

Abstract. Several new characterizations relating the liveness, liveness monotonicity and siphons of general and asymmetric choice nets are first proposed in this article. The first major one states that, if a pure ordinary net satisfies liveness monotonicity, then every siphon of it contains at least one trap. The second major one characterizes the non-liveness of an individual transition of a homogeneous asymmetric choice net in terms of some properties of its siphons. Then, based on these new characterizations, extension and new proofs for several existing characterizations of asymmetric choice nets are given.

Keywords: Asymmetric choice net, characterization, homogeneous, live, liveness monotonicity, Petri net, siphon.

1 Introduction

Verification is the process of showing whether a system has certain properties or not. Techniques for verifying Petri nets can be roughly classified into three approaches: reachability analysis, transformation and characterization. Reachability analysis requires an exhaustive exploration of the state space. The 'state explosion' problem often hinders its applicability to large systems [1]. A transformation aims at simplifying the verification process by restructuring a system by means of reductions, compositions, refinements, modifications, etc. However, especially for general Petri nets, it is usually not easy to find a suitable transformation that will preserve the properties under study [2, 3, 4, 5]. A characterization is a relationship between different proper-

* This research was financially supported by the Research Grant Council of Hong Kong under Grant CityU 1133/99E and CityU 1072/00E and the National Natural Science Foundation of China under Grant No. 60073013.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 203-216, 2002.
© Springer-Verlag Berlin Heidelberg 2002

ties of a system [6]. A relationship may be expressed by means of graph theory, algebra or mathematical programming [7, 8]. It is hoped that, through such a relationship, validity of one property follows from some existing results for the related properties. In the past decades, many characterization results have appeared in the literature [9, 10, 11, 12, 13, 14].

Among others, liveness is undoubtedly one of the most important properties of many systems. It implies the absence of global and local deadlocks. It is one of the fundamental requirements of a system. However, it is also one of the properties most difficult to verify. Very often, direct (i.e., based on its definition) verification is very difficult and characterization is an alternative approach. For general Petri nets, some characterizations for deadlocks have been reported [9]. At present, most of the results for liveness are for special classes of Petri nets and are related to siphons [10, 11, 12, 13, 15], such as free choice nets and extended non self-controlling nets. In fact, the following characterizations for the liveness of asymmetric choice (AC) nets in terms of siphons have been reported: (1) A sufficient condition: An ordinary AC net is live if every siphon contains at least one trap marked by the initial marking [1]. (2) A necessary and sufficient condition [11]: A homogeneous AC (HAC) net is live if and only if, for every marking reachable from the initial marking, every siphon contains a place whose marked value is not less the minimum weight of the outgoing arcs from the place. This characterization is for the entire HAC net. [16] extends such a result to a subset of transitions. We do not know of any characterization for the liveness of individual transitions of HAC nets. Such a result would be very desirable.

Liveness monotonicity is a property related to liveness. It states that, if a Petri net is live with respect to an initial marking, it is still live when the values of the initial marking are increased. If such a property is true, it will save a lot of efforts when checking for a large range of marking values. It has been shown that free choice nets [10] satisfy this property and that an ordinary AC net satisfies liveness monotonicity if and only if every siphon contains a trap marked by the initial marking [12]. Issues concerning liveness-monotonicity limited to a subset of places have also been studied [17]. It is shown there that liveness monotonicity can be applied for considering the preservation of liveness when two nets are joined over a subset of places.

This article generalizes some of the existing results mentioned above in several directions. First, in Section 3, an existing characterization on liveness monotonicity is generalized from AC nets to ordinary nets. In Section 4, a characterization for the non-liveness of an individual transition of an HAC net is proposed. Based on this basic result, several other characterizations on the liveness of HAC nets are derived. Section 5 extends and provides new proofs for some exiting characterizations on AC nets. Some discussion on the main results and possible future work are given in Section 6.

2 Preliminaries

This section provides some fundamentals of Petri nets required for the rest of the article.

A *net* is denoted by $N = (P, T, F, W)$, where P is a finite set of places, T is a finite set of transitions with $P \cap T = \varnothing$, $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation and W is a

weight function on the arcs, i.e., W: $F \to \{1, 2, 3, \ldots\}$. A net N is said to be *ordinary* and is denoted as $N = (P, T, F)$ if every weight is 1. The weight W is said to be *homogeneous* if and only if, $\forall p \in P$, $\forall t_1, t_2 \in p^\bullet$, $W(p, t_1) = W(p, t_2)$. The *pre-set* ${}^\bullet x$ of x is defined as ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and the *post-set* $x^\bullet$ is defined as $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. Similarly, for any subset of $Y \subseteq P \cup T$, ${}^\bullet Y$ (resp., $Y^\bullet$) denotes the union of ${}^\bullet y$ (resp., $y^\bullet$) for all $y \in Y$.

A *marking* of a net $N = (P, T, F, W)$ is a mapping M: $P \to \{0, 1, 2, \ldots\}$. A *Petri net* is a couple (N, M_0), where N is a net and M_0 is the *initial marking* of N. A place p is said to be *marked* by M if $M(p) > 0$. For $P' \subseteq P$, P' is said to be marked by M if $\exists p \in P'$ such that $M(p) > 0$. A transition t is *enabled* or *firable* at a marking M if $M(p) \geq W(p, t)$ $\forall p \in {}^\bullet t$. A transition t may be *fired* if it is enabled. Firing transition t results in changing marking M to a new marking M', where $M'(p) = M(p) - W(p, t) + W(t, p)$ $\forall p \in P$. The firing process is denoted by $M[N, t\rangle M'$. A firing sequence σ is a sequence of consecutively firable transitions. $M[N, \sigma\rangle M'$ means that M' is reachable from M by firing sequence σ. $R(N, M)$ denotes the set of all markings reachable from marking M.

A net $N = (P, T, F, W)$ is said to be *pure* or *self-loop-free* if ${}^\bullet x \cap x^\bullet = \varnothing$ for $\forall x \in P \cup T$. In this paper, all nets are assumed to be pure and connected.

Definition 2.1. Let (N, M_0) be a Petri net.

1) A transition t is said to be *live* if and only if, $\forall M \in R(N, M_0)$, there exists an $M' \in R(N, M)$ such that t can be fired at M'.
2) (N, M_0) is said to be *live* if and only if every transition of N is live.

Definition 2.2. Let $N = (P, T, F)$ be an ordinary net.

1) N is said to be *Free Choice* (*FC*) if and only if, $\forall p_1, p_2 \in P$, $p_1^\bullet \cap p_2^\bullet \neq \varnothing \Rightarrow p_1^\bullet = p_2^\bullet$.
2) N is said to be *Asymmetric Choice* (*AC*) if and only if, $\forall p_1, p_2 \in P$, $p_1^\bullet \cap p_2^\bullet \neq \varnothing \Rightarrow p_1^\bullet \subseteq p_2^\bullet$ or $p_2^\bullet \subseteq p_1^\bullet$.

Definition 2.3. A net $N = (P, T, F, W)$ is said to be *Homogeneous Asymmetric Choice* (*HAC*) if and only if N is AC and W is homogeneous.

Definition 2.4. A set of places D is called a *siphon* if ${}^\bullet D \subseteq D^\bullet$. D is called a *trap* if $D^\bullet \subseteq {}^\bullet D$. A siphon (resp., trap) is said to be *minimal* if it does not properly contain other siphons (resp., traps). A siphon (resp., trap) is said to be *maximal* if and only if it is not contained in another siphon (resp., trap) which is not P (i.e., the set of all places).

It is well known that a siphon remains token-free once it becomes free of tokens and a trap remains marked once it becomes marked.

3 Characterizing Liveness Monotonicity of Petri Nets

This section presents a new characterization concerning the liveness monotonicity of ordinary Petri nets in terms of siphons. Liveness monotonicity is the property that liveness is preserved if tokens are added in some of the places of the initial marking.

Definition 3.1. A net N is said to satisfy the *Siphon-trap Property* if every siphon of N contains at least one trap.

Definition 3.2. A net N is said to satisfy *liveness monotonicity* at M_0 if (N, M) is live for any M such that $M \geq M_0$. N satisfies liveness monotonicity if there exists M_0 such that N satisfies liveness monotonicity at M_0.

In general, an ordinary net does not always satisfy liveness monotonicity if some of its siphons does not contain any trap. For example, the Petri net (N, M_0) in Fig. 1 is live. $D = \{p_1, p_2, p_3, p_4\}$ is a siphon of N that does not contain any trap. However, (N, M) is not live for $M = (0\ 0\ 0\ 1\ 1) \geq M_0 = (0\ 0\ 0\ 1\ 0)$ since D will be emptied after firing t_5 from M.

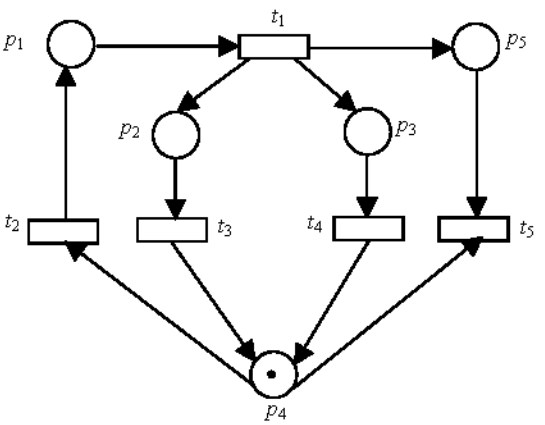


Fig. 1. A live Petri net that does not satisfy liveness monotonicity

As far as we know, the following necessary condition is a new characterization for the liveness monotonicity of ordinary nets.

Theorem 3.1. A net N satisfies the Siphon-trap Property if N satisfies liveness monotonicity.

Proof. (By contradiction) Suppose N has a siphon D that does not contain any trap and (N, M_0) is live for some M_0. The following process described in pseudo code creates a marking M such that $M \geq M_0$ and $M(p) > M_0(p)$ for at least one $p \in P$ and a firing sequence σ such that firing σ from M will take away all tokens from D. Hence, (N, M) is not live.

Process:

Step 1. To create a set of firable transitions $\{t_i\}$ from $D^\bullet$ and their presets $\{I_i\}$ from D:

Initially, $i := 0$; $D_i := D$

while $D_i \neq \varnothing$

$i := i + 1$;

Select $t_i \in (D_{i-1})^\bullet$ such that $t_i^\bullet \cap D_{i-1} = \varnothing$; (1)

$I_i := {}^\bullet t_i \cap D_{i-1}$; $D_i := D_{i-1} - I_i$; (2)

end while

$n := i$.

/* n is the maximum number of transitions to be fired in Step 2 */

Step 2. To create a marking M and a firing sequence σ based on $\{t_i\}$ and $\{I_i\}$:

Initially, $i := n$; $M := M_0$; $M' := M_0$

/* M is the marking to be created so that (N, M) is not live. M' is the intermediate marking to be updated in the process of creating M. */

while $i > 0$

$m_i := \max\{M'(p) \mid p \in I_i\}$

if $m_i > 0$ **then**

begin

for every $p \in {}^\bullet t_i$, **if** $M'(p) < m_i$ **then**

begin $M(p) := M(p) + m_i - M'(p)$;

$M'(p) := m_i$ **end** (3)

/* To make t_i firable m_i times by modifying M' */

$\sigma := \sigma t_i \ldots t_i$

/* t_i is fired m_i times consecutively */

for every $p \in {}^\bullet t_i$: $M'(p) := M'(p) - m_i$;

for every $p \in (t_i)^\bullet$: $M'(p) := M'(p) + m_i$

/* To update M' after firing t_i m_i times*/

end begin

i := i – 1

end while

Validity of this theorem is a consequence of the following properties of the above process:

a) Step 1 will terminate.
Since $t_i \in (D_{i-1})^\bullet$, $I_i = {}^\bullet t_i \cap D_{i-1} \neq \emptyset$ for $1 \leq i \leq n$. Since $D_i = D_{i-1} - I_i$ from (2), $D = D_0 \supset D_1 \supset D_2 \supset \ldots \supset D_n$. Hence, D_i will eventually become empty and Step 1 will terminate.

b) If $D_i \neq \emptyset$, a transition t satisfying (1) can always be found.
If, for every $t \in D_i{}^\bullet$, $t^\bullet \cap D_i \neq \emptyset$, then $D_i{}^\bullet \subseteq {}^\bullet D_i$, i.e., D_i is a trap of D because $D_i \subseteq D$ as shown above, contradicting with the fact that D does not contain any trap.

c) $M \geq M_0$ and $M(p) > M_0(p)$ for at least one $p \in P$.
This follows from (3) within Step 2.

d) is firable at M and D will be emptied after firing σ.
Since $I_i \neq \emptyset$ and $D_i = D_{i-1} - I_i$, $D_i \subset D_{i-1}$. Hence, $D_k \subset D_{i-1}$ $\forall k \geq i$. If $t_i^\bullet \cap D_{i-1} = \emptyset$, $t_i^\bullet \cap I_k = t_i^\bullet \cap ({}^\bullet t_k \cap D_k) \subseteq t_i^\bullet \cap D_{i-1} = \emptyset$ $\forall k > i$. This implies that firing t_i in Step 2 does not put tokens into I_k $\forall k \geq i$. Since $t_i{}^\bullet \cap D_{i-1} = \emptyset$, after t_i is fired m_i times, every place of I_i will lose m_i tokens. Let σ' be the part of σ which contains $m_n\ t_n$, …, and $m_{i+1}\ t_{i+1}$. Suppose M' is the marking reachable from M after firing σ'. If ${}^\bullet t_i \cap I_k \neq \emptyset$ for some $k > i$, then, according to (3), there are enough tokens in I_k such that t_i can be fired m_i times. After σ being fired at M, all tokens in I_n, I_{n-1}, …, I_1 will be taken away. Since $I_n = D_{n-1}$, $D = D_0 = D_1 \cup I_1 = \ldots = \cup_{i=1\ldots n} I$. This implies that D will be emptied.

We have shown that there exists M' such that $M[N, \sigma\rangle M'$, $M'(D) = \emptyset$ and no $t \in D^\bullet$ can be enabled at M'. Hence, (N, M) is not live. □

The following example illustrates the rather complex proof of Theorem 3.1.

Example. Consider the Petri net (N, M_0) in Fig. 2, where $M_0 = (3, 1, 0, 1, 5, 1)$. Siphon $D = \{p_1, p_2, p_3, p_4\}$ does not contain any trap.

Step 1: To create a set of firable transitions $\{t_i\}$ from $D^\bullet$ and their presets $\{I_i\}$ from D.

Initially, $D_0 = D = \{p_1, p_2, p_3, p_4\}$.

For iteration $i = 1$, select $t_1 \in D_0^\bullet = \{t_1, t_2, t_3, t_4, t_5\}$, where $t_1^\bullet \cap D_0 = \varnothing$. Let $I_1 = {}^\bullet t_1 \cap D_0 = \{p_2, p_3\}$ and $D_1 = D_0 - I_1 = \{p_1, p_4\}$.

For iteration $i = 2$, select $t_5 \in D_1^\bullet = \{t_3, t_5\}$, where $t_5^\bullet \cap D_1 = \varnothing$. Let $I_2 = {}^\bullet t_5 \cap D_1 = \{p_4\}$ and $D_2 = D_1 - I_2 = \{p_1\}$.

For iteration $i = 3$, select $t_3 \in D_2^\bullet = \{t_3\}$, where $t_3^\bullet \cap D_2 = \varnothing$. Let $I_3 = {}^\bullet t_3 \cap D_2 = \{p_1\}$ and $D_3 = D_2 - I_3 = \varnothing$.

Since $D_3 = \varnothing$, Step 1 terminates with $n = 3$. The set of firable transitions $\{t_1, t_5, t_3\}$ is created with presets I_1, I_2 and I_3, respectively.

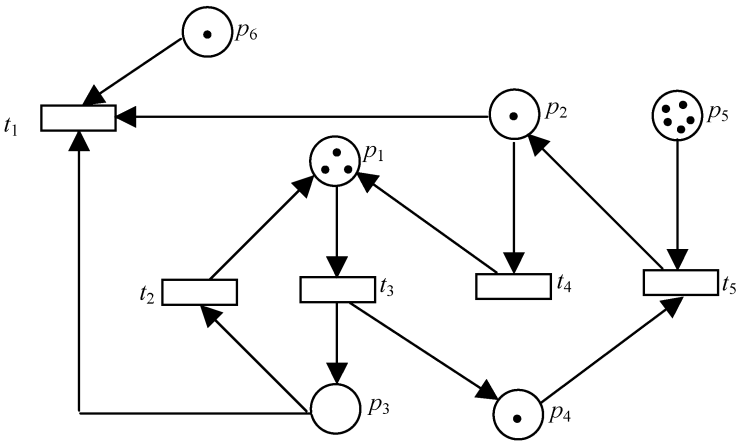


Fig. 2. A Petri net illustrating the constructive proof of Theorem 3.1

Step 2: To create a marking M and a firing sequence σ.

Initially, $i = n = 3$ and $M = M' = M_0 = (3, 1, 0, 1, 5, 1)$.

For $i = 3$, $I_3 = \{p_1\}$, $m_3 = 3$ and ${}^\bullet t_3 = \{p_1\}$. Since $M'(p_1) = m_3$, t_3 can be fired without modifying M and M'. Hence, $M = (3, 1, 0, 1, 5, 1)$ and $M' = (3, 1, 0, 1, 5, 1)$. After firing t_3 3 times, $\sigma = t_3t_3t_3$ and M' becomes (0, 1, 3, 4, 5, 1).

For $i = 2$, $I_2 = \{p_4\}$, $m_2 = 4$, ${}^\bullet t_5 = \{p_4, p_5\}$. Since $M'(p_4) = m_2$ and $M'(p_5) = 5 > m_2$, t_3 can be fired without modifying M and M'. Hence, $M = (3, 1, 0, 1, 5, 1)$ and $M' = (0, 1, 3, 4, 5, 1)$. After firing t_5 4 times, $\sigma = t_3t_3t_3t_5t_5t_5t_5$ and M' becomes (0, 5, 3, 0, 1, 1).

For $i = 1$, $I_1 = \{p_2, p_3\}$, $m_1 = 5$, ${}^\bullet t_1 = \{p_2, p_3, p_6\}$. $m_1 = M'(p_2)$, $M'(p_3) = 3 < m_1$ and $M'(p_6) = 1 < m_1$, t_1 can be fired m_i times only by changing M to (3, 1, 2, 1, 5, 5) and M' to (0, 5, 5, 0, 1, 5). After firing t_1 5 times, $\sigma = t_3t_3t_3t_5t_5t_5t_5t_1t_1t_1t_1t_1$ and M' becomes (0, 0, 0, 0, 1, 0).

When the process terminates, the marking $M = (3, 1, 2, 1, 5, 5)$ and the firing sequence $\sigma = t_3t_3t_3t_5t_5t_5t_5t_1t_1t_1t_1t_1$ are created. After firing σ from M, the marking $M' = (0, 0, 0, 0, 1, 0)$ is reached and $M'(D) = 0$. □

4 Characterizing Liveness of HAC Nets in Terms of Siphons

This section proposes a characterization (Theorem 4.1) for the non-liveness of the individual transitions of an HAC. It states that, for an HAC net, if a transition t is not live, then there must exist a siphon D and a reachable marking M such that ${}^\bullet t \cap D \neq \varnothing$ and, for every $p \in D$, $M(p) < W(p, t') \; \forall t' \in p^\bullet$. While there exists a characterization for the liveness of an HAC net as a whole [11], we think that this is a new result for individual transitions. We also show that the existing characterization follows easily from our result.

Our result extends an existing similar characterization from ordinary AC nets [16] to HAC nets. Note that, as shown in the following example, this characterization requires both homogeneity in the net's weights and asymmetric free-choice in the net's structure.

Example. This example shows that the characterization mentioned above is not valid for homogeneous non-AC nets. Consider the net N in Fig. 3 N is homogeneous but not AC because of $(p_3)^\bullet \cap (p_4)^\bullet \neq \varnothing$ but $(p_3)^\bullet \not\subset (p_4)^\bullet$ and $(p_4)^\bullet \not\subset (p_3)^\bullet$. t_1 is non-live. $D_1 = \{p_1, p_2, p_3\}$, $D_2 = \{p_1, p_3, p_4\}$ and $D_3 = \{p_1, p_2, p_3, p_4\}$ are all siphons such that ${}^\bullet t_1 \cap D_i \neq \varnothing$, $i = 1, 2, 3$. However, for any marking M reachable from the initial marking (1 0 0 0), there exist $p \in D_i$ such that $M(p) \geq W(p, t') \; \forall t' \in p^\bullet$.

This characterization is not valid for weighted but inhomogeneous AC nets either. For example, for the net N of Fig. 4, $R(N, M_0) = \{M_0, M_1\}$, where $M_0 = (2\ 0)$ and $M_1 = (0\ 1)$. $\{p_1, p_2\}$ is the only siphon of N and t_1 is the only non-live transition of (N, M_0). However, $M_0(p_1) = W(p_1, t_2) = 2$ and $M_1(p_2) = W(p_2, t_3) = 1$.

Lemma 4.1. Let (N, M_0) be an HAC net, where $N = (P, T, F, W)$. If t is the only non-live transition of (N, M_0), then there exist $p \in {}^\bullet t$ and $M \in R(N, M_0)$ such that $\{p\}$ is a siphon and $M(p) < W(p, t)$.

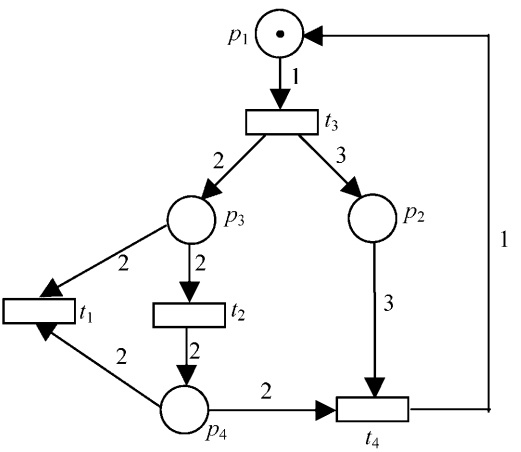


Fig. 3. A marked Petri net with non-live transition t_1

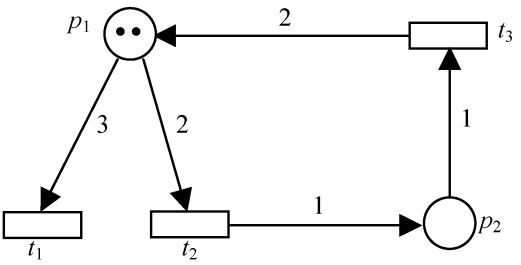


Fig. 4. An inhomogeneous AC net with non-live transition t_1

Proof. Obviously, ${}^\bullet t \neq \varnothing$ because, otherwise, t is live. In the following, we are going to show that $\exists p \in {}^\bullet t$ such that ${}^\bullet p = \varnothing$. If this is true, then ${}^\bullet p \subseteq p^\bullet$ and $\{p\}$ is the siphon being sought. Then, by firing those transitions in $p^\bullet$ (including t) as many times as necessary, a marking M will be reached such that $M(p) < W(p, t)$.

It will be shown that contradiction will arise if ${}^\bullet p \neq \varnothing$ for every $p \in {}^\bullet t$. Let ${}^\bullet t$ be partitioned into two sets (Fig. 5): $H = \{p \mid p \in {}^\bullet t \text{ and } p^\bullet = \{t\}\}$ and $K = \{p \mid p \in {}^\bullet t \text{ and } p^\bullet \supset \{t\}\}$.

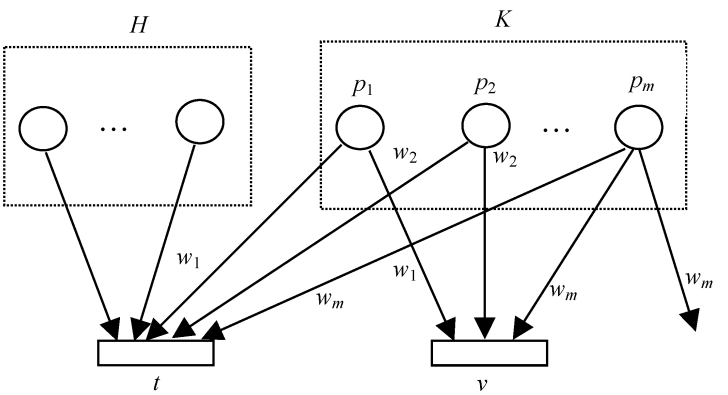


Fig. 5. Explanation for the proof of Lemma 4.1

Since H and K cannot be both empty, one or both of the following cases are true.

Case 1 ($H \neq \varnothing$): Every $p \in H$ has the following characteristic: Since t is non-live, there exists $M \in R(N, M_0)$ such that t cannot be enabled at any $M' \in R(N, M)$. That is, at and after M, the number of tokens in p can never be deducted. On the other hand, since ${}^\bullet p \neq \varnothing$, $t \notin {}^\bullet p$ (because N has no self-loops) and t is the only non-live transition, every transition in ${}^\bullet p$ is live and can be fired to increase the tokens in p. Hence, there will eventually be a marking $M' \in R(N, M)$ such that $M'(p) \geq W(p, t)\ \forall p \in H$. Also, $\forall M'' \in R(N, M'), M''(p) \geq W(p, t)\ \forall p \in H$.

Case 2 ($K \neq \varnothing$): Let $K = \{p_1, p_2, \ldots, p_m\}$, where $m \geq 1$. Since N is an AC net, without loss of generality, we may assume that $p_1^\bullet \subseteq p_2^\bullet \subseteq \ldots \subseteq p_m^\bullet$. Since $p_1^\bullet - \{t\} \neq \varnothing$ by the definition of K, $\exists v \in p_1^\bullet - \{t\}$. Then, $v \in p_i^\bullet$, for $i = 1, \ldots, m$. This implies

that $K \subseteq {}^\bullet v$. Since v is live, at the M' obtained in Case 1, there exists a firing sequence σ not including t and a marking $M'' \in R(N, M')$ at which v is firable. That is, $M''(p) \geq W(p, v)\ \forall p \in K$. Since $K \subseteq {}^\bullet t$ and W is homogeneous, $W(p, t) = W(p, v)\ \forall p \in K$. This implies that $M''(p) \geq W(p, t)\ \forall p \in K$.

Combining Case 1 and Case 2, we have $M''(p) \geq W(p, t)\ \forall p \in H \cup K = {}^\bullet t$, implying that t can be enabled at M'' – a contradiction. □

Theorem 4.1. Let (N, M_0) be an HAC net, where $N = (P, T, F, W)$. Then, $t \in T$ is non-live in (N, M_0) iff there exist a siphon D and a marking $M \in R(N, M_0)$ such that (1) ${}^\bullet t \cap D \neq \emptyset$; (2) $\forall p \in D$: $M(p) < W(p, t')\ \forall t' \in p^\bullet$.

Proof. "⇐" Condition (2) implies that there exists a marking $M \in R(N, M_0)$ at which, for every $p \in D$: $M(p) < W(p, t')\ \forall t' \in p^\bullet$. That is, no $t \in D^\bullet$ can be enabled at M. Since ${}^\bullet D \subseteq D^\bullet$, no $t \in D^\bullet$ can be enabled and the number of tokens in any $p \in D$ remains unchanged. Hence, $M'(p) = M(p) < W(p, t')\ \forall t' \in p^\bullet$ for every $M' \in R(N, M)$ and every $p \in D$. This means that t cannot be enabled at M', i.e., t is non-live in (N, M_0).

"⇒" (By induction on $|T|$) For $|T| = 1$, i.e., t is the only non-live transition in (N, M_0). The conclusion follows from Lemma 4.1. Assume that, the conclusion is true whenever $|T| \leq m$. For $|T| = m + 1$, consider the following two cases:

Case 1. Every $v \in T - \{t\}$ is live in (N, M_0).
Since t is the only non-live transition in (N, M_0), the conclusion follows from Lemma 4.1.

Case 2. There exist other transitions that are not live in (N, M_0).
Since t is not live in (N, M_0), there exists $M \in R(N, M_0)$ such that t cannot be enabled at M' for any $M' \in R(N, M)$. Consider two subcases for the net (N, M):

(1) t is the only non-live transition in (N, M).
By Lemma 4.1, there exist $p \in {}^\bullet t$ and $M' \in R(N, M)$ such that $\{p\}$ is a siphon and $M'(p) < W(p, t)$.

(2) $\exists u \in T - \{t\}$ that is also not live in (N, M).
Since u is non-live in (N, M), there exists $M_1 \in R(N, M)$ such that u cannot be enabled at M_2 for any $M_2 \in R(N, M_1)$. This means that t and u are both not live in (N, M_1).

Let N_t be the net after deleting t and all its associated arcs in N. It is obvious that N_t is still an HAC net. Since u cannot be enabled at any marking reachable from M_1, u is not live in (N_t, M_1). Since the number of transitions of N_t is m, there exist a siphon D_u in N_t satisfying ${}^\bullet u \cap D_u \neq \emptyset$ and $M_u \in R(N_t, M_1)$ such that $\forall p \in D_u$: $M_u(p) < W(p, t')\ \forall t' \in p^\bullet$. Similarly, t is not live in (N_u, M_u), where N_u is the net after deleting u and all its associated arcs in N. Hence, there exist a siphon D_t in N_u satisfying ${}^\bullet t \cap D_t \neq \emptyset$ and $M_t \in R(N_u, M_u)$ such that $\forall p \in D_t$: $M_t(p) < W(p, t')\ \forall t' \in p^\bullet$.

Since $M_t \in R(N_t, M_u)$, for every $p \in D_u$: $M_t(p) < W(p, t')\ \forall t' \in p^\bullet$. This implies that $M_t(p) < W(p, t')\ \forall p \in (D_t \cup D_u)$ and $\forall t' \in p^\bullet$. In N, since $t \in D_t^\bullet$ and $u \in D_u^\bullet$, ${}^\bullet(D_t \cup D_u) \subseteq {}^\bullet D_t \cup {}^\bullet D_u \cup \{u, t\} \subseteq D_t^\bullet \cup D_u^\bullet = (D_t \cup D_u)^\bullet$, i.e., $D_t \cup D_u$ is a siphon of N. It is

obvious that $R(N, M_1) = R(N_t, M_1)$ and $R(N, M_u) = R(N_u, M_u)$. Hence, $M_t \in R(N, M_u) \subseteq R(N, M_1) \subseteq R(N, M) \subseteq R(N, M_0)$. □

Characterization for the liveness of an entire HAC net, already proved in [11], follows easily from our above result.

Corollary 4.1. [11] An HAC net (N, M_0) is live iff, for every (minimal) siphon D of N and for every marking $M \in R(N, M_0)$, there exists some $p \in D$ such that $M(p) \geq W(p,t)$.

Proof. "⇒" For an arbitrary (minimal) siphon D. There must exist t such that $t \in D^\bullet$, i.e., ${}^\bullet t \cap D \neq \varnothing$. Since t is live in (N, M_0), it follows from Theorem 4.1 that, for every marking $M \in R(N, M_0)$, there exists some $p \in D$ such that $M(p) \geq W(p, t')\ \forall t' \in p^\bullet$. "⇐" It is obvious. □

Based on Theorem 4.1, we can derive another characterization for the liveness of an HAC net in terms of the liveness of those subnets generated by its siphons.

Definition 4.1 (N_D). Let (N, M_0) be a general net, where $N = (P, T, F, W)$. For $D \subseteq P$, (N_D, M_D) is called a *D-induced subnet* of (N, M_0), where $N_D = (D, D^\bullet, F_D, W_D)$, $F_D = F \cap ((D \times D^\bullet) \cup (D^\bullet \times D))$, $W_D = W \mid F_D$ and $M_D = M_0 \mid D$. (For $D \subseteq P$, $M \mid D$ denotes the *restriction* (i.e., subvector) of M over the set D.)

Corollary 4.2. Let (N, M_0) be an HAC net, where $N = (P, T, F, W)$. If, for every (minimal) siphon D of N, the D-induced subnet (N_D, M_D) is live, then (N, M_0) is live.

Proof. Suppose $\exists t$ which is not live in (N, M_0). By Theorem 4.1, there exist a siphon D, a transition sequence $\sigma \in T^*$ and $M' \in R(N, M_0)$ such that $t \in D^\bullet$, $M_0[N, \sigma\rangle M'$ and $\forall p \in D$: $M'(p) < W(p, t')\ \forall t' \in p^\bullet$. Let σ_D be the restriction of σ over $D^\bullet$. Since ${}^\bullet D \subseteq D^\bullet$, the firing of any transition in $T \backslash D^\bullet$ does not influence the token distribution in D. Hence, σ_D can also be fired in (N_D, M_D), with the result that $M_D[N_D, \sigma_D\rangle M_D'$ and $\forall p \in D$: $M_D'(p) < W(p, t')\ \forall t' \in p^\bullet$. By Theorem 4.1, t is not live in (N_D, M_D), contradicting with the fact that (N_D, M_D) is live. □

To determine the liveness of an HAC net, Corollary 4.2 checks the liveness of the *D-induced subnet* for every (minimal) siphon D. In practice, many Petri nets have a large number of (minimal) siphons. As shown below, it is sufficient just to check a much smaller set of siphons, the set of maximal siphons, i.e., siphons not properly contained in another siphon which does not include all places of the net.

Corollary 4.3. Let (N, M_0) be an HAC net, where $N = (P, T, F, W)$. If the D-induced subnet (N_D, M_D) is live for every maximal siphon D of N, then (N, M_0) is live.

Proof. Suppose $\exists t$ which is not live in (N, M_0). By Theorem 4.1, there exist a siphon D' satisfying $t \in (D')^\bullet$, a transition sequence $\sigma \in T^*$ and $M' \in R(N, M_0)$ such that $M_0[N, \sigma\rangle M'$ and $\forall p \in D'$: $M'(p) < W(p, t')\ \forall t' \in p^\bullet$. Since there exists a maximal siphon D which contains D', let σ_D be the restriction of σ over $D^\bullet$. Since ${}^\bullet D \subseteq D^\bullet$, the firing of any transition in $T \backslash D^\bullet$ does not influence the token distribution in D. Hence, σ_D can also be fired in (N_D, M_D), with the result that $M_D[N_D, \sigma_D\rangle M_D'$ and $\forall p \in D'$: $M_D'(p) < W(p, t')\ \forall t' \in p^\bullet$. By Theorem 4.1, t is not live in (N_D, M_D), contradicting with the fact that (N_D, M_D) is live. □

Example. (Illustration for Corollaries 4.2 and 4.3.) (Fig. 6) Note that the sufficient condition for the liveness of an HAC net as stated in Corollary 4.2 is not a necessary condition. For example, for the live AC net in Fig. 6, the D-induced subnet (N_D, M_D) for the minimal siphon $D = \{p_1, p_2, p_3, p_4\}$ is not live. On the other hand, consider the only two maximal siphons $D_1 = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ and $D_2 = \{p_2, p_3, p_4, p_5, p_6, p_7\}$ in the HAC net. (Remember that, by the definition of a maximum siphon, the only siphon larger than D_1 and D_2 is P itself.) Since both (N_{D1}, M_{D1}) and (N_{D2}, M_{D2}) are live, it follows from Corollary 4.3 that (N, M_0) is live.

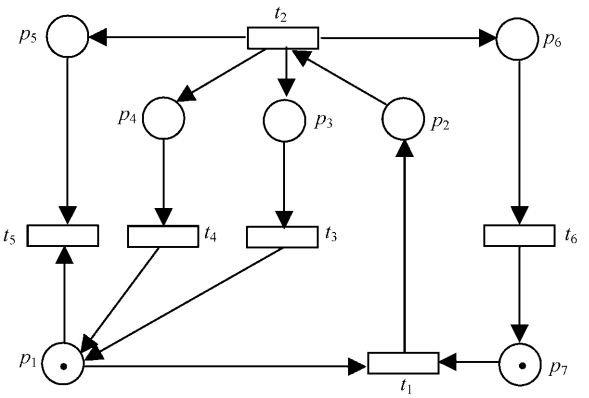


Fig. 6. A live AC net (N, M_0)

5 Characterizations for Ordinary AC Nets

This section first extends and modifies an existing characterization for minimal siphons from FC nets [19] to AC nets (Theorem 5.1). Then, based on this extension, it provides new proofs for a well-known characterization for the liveness of AC nets (Corollary 5.1 [1]) and two less-known characterizations (Corollary 5.2 [18] and Corollary 5.3 [12]) for the liveness monotonicity of AC nets.

In general, it is time-consuming to determine whether a siphon is minimal or not. In the literature, there exist several results characterizing the minimality of a siphon in terms of their induced nets. In particular, it is shown in [19] that, for an FC net $N = (P, T, F)$ and a siphon D of N, D is minimal iff (1) the induced subnet $N_D = (D, {}^\bullet D, D^\bullet)$ is strongly connected, and (2) $|{}^\bullet t \cap D| \leq 1$ for every $t \in D^\bullet$. In Theorem 5.1 below, we extend this result to ordinary AC nets. Furthermore, there is a slight modification in the result: While [1] characterizes the minimality of a set of places known to form a siphon, our result characterizes an arbitrary set of places for being a minimal siphon. Let us first state two known results below.

Property 5.1. [20] For a minimal siphon D of net N, the induced subnet subnet $N_D = (D, {}^\bullet D, D^\bullet)$ is strongly connected.

Property 5.2. [1] An FC net (N, M_0) is live iff M_0 marks a trap of every siphon.

Theorem 5.1. For an AC net $N = (P, T, F)$, let $D \subseteq P$ and $N_D = (D, {}^\bullet D, D^\bullet F)$ be the induced subnet. Then, D is a minimal siphon iff (1) N_D is strongly connected, and (2) $|{}^\bullet t \cap D| = 1$ for every $t \in D^\bullet$.

Proof. "⇐" Since N_D is strongly connected, for any $t \in {}^\bullet D$, there exists $p \in D$ such that $t \in p^\bullet$. This implies that $t \in D^\bullet$ and D is a siphon. For any siphon D' within D, let $p \in D$, $p' \in D'$ and $t \in {}^\bullet D$ such that the arcs (p, t) and (t, p') belong to ${}^\bullet F$. This is possible because N_D is strongly connected. Then, since $t \in {}^\bullet p' \subseteq {}^\bullet D' \subseteq (D')^\bullet \subseteq D^\bullet$, it follows from Condition (2) that p is the only input place of t in the entire D, and in particular, in D'. This requires p to belong to D' also. By repeating the same argument, it follows that $D' = D$ and D is minimal.

"⇒" Suppose D is a minimal siphon. Condition (1) follows from Property 5.1. Suppose there exists $t \in D^\bullet$ such that ${}^\bullet t \cap D = \{p_1, p_2, \ldots, p_m\}$, where $m \geq 2$. Since N is an AC net, without loss of generality, we can assume that $p_1^\bullet \subseteq p_2^\bullet \subseteq \ldots \subseteq p_m^\bullet$. Since $p_m \in D - \{p_1\}$ and $p_1^\bullet \subseteq p_m^\bullet$, deleting p_1 from D does not reduce $D^\bullet$, i.e., $D^\bullet = (D - \{p_1\})^\bullet$. Hence, ${}^\bullet(D - \{p_1\}) \subseteq {}^\bullet D \subseteq D^\bullet = (D - \{p_1\})^\bullet$, i.e., $D - \{p_1\}$ is a siphon. This contradicts with the fact that D is minimal. □

Based on Theorem 5.1, we provide a new proof for the well-known result below.

Corollary 5.1. [1] An ordinary AC net (N, M_0) is live if every siphon of N contains a trap marked by M_0.

Proof. Let D be an arbitrary minimal siphon of N and (N_D, M_D) be the D-induced subnet. Since M_0 marks a trap of every siphon of N, M_D marks a trap of D. Hence, M_D marks a trap of every siphon of (N_D, M_D) because D is the only siphon of N_D. By Theorem 5.1, $|{}^\bullet t \cap D| = 1$ for every $t \in D^\bullet$. This means that (N_D, M_D) is in fact an FC net and is live according to Property 5.2. By Corollary 4.2, (N, M_0) is live. □

Two recent results [12, 18] concerning the liveness monotonicity of AC nets are not yet well known and their proofs are quite complex. Hence, they are restated as below with new proofs on the basis of Theorem 3.1 and Theorem 4.1.

Corollary 5.2. [18] An AC net N satisfies the Siphon-trap Property iff N satisfies liveness monotonicity.

Proof. "⇒" Let M_0 be a marking which marks at least one place of each of these traps. By Corollary 5.1, (N, M_0) is live. For any $M \geq M_0$, these traps are also marked by M. By Corollary 5.1 again, (N, M) is live. "⇐" This follows from Theorem 3.1. □

Corollary 5.3. [12] An AC net N satisfies liveness monotonicity at M_0 iff M_0 marks a trap of every minimal siphon.

Proof. "⇒" Suppose M_0 does not mark any trap of some minimal siphon D. By Theorem 4.1, $|{}^\bullet t \cap D| = 1 \ \forall t \in D^\bullet$. This means that the D-induced subnet (N_D, M_D) is an FC net. Since any trap of D is unmarked by M_D, (N_D, M_D) is not live by Property 5.2. Suppose t is a non-live transition in (N_D, M_D). According to Theorem 4.1, there exist a siphon D' satisfying ${}^\bullet t \cap D' \neq \varnothing$, a marking M_1 and a firing sequence $\sigma \in D^*$ such that $M_D[N_D, \sigma\rangle M_1$ and $M_1(D') = 0$. Since D is minimal, $D' = D$. Putting enough tokens to ${}^\bullet T_1 - D$, where $T_1 = \{t \mid t \in \sigma\}$, such that a new marking $M \geq M_0$ can be obtained and

σ can be fired at M. Since $M(p) = M_D(p)$ for every $p \in D$, there exists M'' such that $M[N, \sigma\rangle M''$ and $M''(D) = 0$. According to Theorem 4.1, (N, M) is not live.

"$\Leftarrow$" Suppose M_0 marks a trap of every minimal siphon. For every $M \geq M_0$, M also marks a trap of every minimal siphon. By Corollary 5.1, (N, M) is live. □

6 A Concluding Remark

Characterization in terms of siphons has been a major approach for the investigation of liveness and liveness monotonicity of nets. Important results have been obtained for special classes of nets up to AC nets. This article extends some of the results in several directions: (1) characterization on liveness monotonicity is extended from AC nets to ordinary nets. (2) Characterization for the non-liveness of an individual transition is extended from ordinary AC nets to HAC nets. (3) Modifications and new proofs are provided for some of the existing characterizations for AC nets.

It seems natural that, in the next stage for research in the characterization of properties with respective to monotonicity of markings, more attention can be paid on three directions: (1) For the study of liveness monotonicity, to extend from ordinary nets to nets with general weights. This is part of our current research project. (2) To generalize the definition of liveness monotonicity from positive increments to a combination of both positive and negative increments. This generalization can be used to study, for example, the issue of liveness preservation when two nets are composed through fusing a few of their places [17], because, after the composition, the tokens in one of the nets may flow to the other and thus be lost there forever. As a result, the marking of the first net has a negative increment while the second has a positive increment. (3) To study the effects of marking changes on other properties than liveness, such as reversibility, boundedness, etc.

References

1. Murata, T.: Petri Nets: Properties, Analysis, and Applications. Proceedings of the IEEE, Vol. 77, No. 4 (1989) 541-580: The Stanford Digital Library Metadata Architecture. Int. J. Digit. Libr. 1 (1997) 108–121.
2. Esparza, J.: Reduction and Synthesis of Live and Bounded Free Choice Petri Nets. Information and Computation, Vol. 114, No. 1 (1994) 50-87.
3. Mak, W. M.: Verifying Property Preservation for Component-Based Software Systems (A Petri-net Based Methodology). PhD Thesis, Dept. of Computer Science, City University of Hong Kong, June 2001.
4. Suzuki, I. and Murata, T.: A Method for Stepwise Refinement and Abstraction of Petri Nets. Journal of Computer and System Sciences, Vol. 27 (1983) 51-76.
5. Cheung, T. Y. and Zeng, W.: Invariant-preserving Transformations for the Verification of Place/Transition Systems. IEEE Transactions on Systems, Man and Cybernetics – Part A: Systems and Humans, Vol. 28, No.1 (1998) 114-121.
6. Esparza, J. and Silva, M.: On the Analysis and Synthesis of Free Choice Systems. Lecture Notes in Computer Science, Vol. 483. Springer-Verlag, (1990) 243-286.

7. Best, E.: Structure Theory of Petri Nets: the Free Choice Hiatus. Lecture Notes in Computer Science, Vol. 254. Springer-Verlag, (1987) 168-205.
8. Sliva, M., Teruel, E. and Colom, J. M.: Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition Net Systems. Lecture Notes in Computer Science, Vol. 1491. Springer-Verlag, (1998) 309-373.
9. Chu, F. and Xie, X.: Deadlock Analysis of Petri Nets Using Siphon and Mathematical Programming. IEEE Trans. on Robotics and Automation, Vol. 13, No.6 (1997) 793-804.
10. Desel, J. and Esparza, T.: Free Choice Petri Nets. Cambridge University Press (1995).
11. Barkaoui, K. and Pradat-Peyre, J.: On Liveness and Controlled Siphons in Petri Nets. Lecture Notes in Computer Science, Vol. 1091. Springer-Verlag, (1996) 57-72.
12. Zhen, Q. and Lu, W. M.: On Liveness and Safeness of Asymmetric Choice Nets. Journal of Chinese Software, Vol. 11, No.5 (2000) 590-605.
13. Barkaoui, K., Couvreur, J. M. and Dutheiliet C.: On Liveness in Extended Non Self-Controlling Nets. Lecture Notes in Computer Science, Vol. 935. Springer-Verlag, (1995) 25-44.
14. Ridder, H. and Lautenbach, K.: Liveness in Bounded Petri Nets Which are covered by t-invariants. Lecture Notes in Computer Science, Vol. 815. Springer-Verlag, (1994) 358-378.
15. Barkaoui, K. and Minoux, M.: A Polynomial-time Graph Algorithm to Decide Liveness of Some Basic Classes of Bounded Petri Nets. Lecture Notes in Computer Science, Vol. 616. Springer-Verlag, (1992) 62-75.
16. Iordache, M. V. and Antsaklis, P. J.: Generalized Conditions for Liveness Enforcement and Deadlock Prevention in Petri Nets. Lecture Notes in Computer Science, Vol. 2075. Springer-Verlag, (2001) 184-203.
17. Souissi, Y.: On Liveness Preservation by Composition of Nets via a Set of Places. Lecture Notes in Computer Science, Vol. 524. Springer-Verlag, (1991) 277-295.
18. Jiao, L., Lu, W. M. and Cheung, T. Y.: On Liveness Monotonicity of Petri Nets. Proceeding of 2001 International Technical Conference on Circuits/Systems, Computers and Communications, Tokushima, Japan (2001) 311-315.
19. Esparza, J. and Silva, M.: A Polynomial-time Algorithm to Decide Liveness of Bounded Free Choice Nets. Theoretical Computer Science 102 (1992) 185-205.
20. Holk, M.: Analysis of Production Schemata by Petri Nets. TR-94, MIT, Project MAC, Boston 1972, (a second version appeared in 1974).

Petri Nets, Situations, and Automata

Ekkart Kindler*

Institut für Informatik, Technische Universität München
D-80290 München
kindler@informatik.hu-berlin.de

Abstract. Scenarios are now widely used when developing distributed systems. In this paper, we provide a semantical foundation for scenarios, which reflects the informal and non-operational use of scenarios. We call this semantics *situations*. We show that system properties can be specified by implications between situations, and we provide some proof techniques for verifying that a system meets such specifications. The basic proof arguments are *automata*, which are abstractions of the considered system or some of its causal relations.

1 Introduction

Ever since people are building distributed systems, they have been using some kind of graphical representations of the system's executions for understanding, discussing, specifying, and verifying the system's behaviour. In most cases, these graphical representations have been informal rough sketches without any precise semantics. Nevertheless, these rough sketches helped understanding the interoperation of the system's components and helped building correct systems. The principle that makes this procedure work is *abstraction*: The developers did neither consider all executions, nor did they consider complete executions. Rather, they concentrated on the crucial parts of the crucial executions and abstracted from unnecessary details. Therefore, we do not call these graphical representations executions, but we call them *scenarios*.

In the mid-90ties, the use of scenarios has become quite popular for software development due to the success of *Message Sequence Charts* (MSC) [8] and *UML* with its *sequence diagrams* [14]. Much research has been done and is still being done on how to use scenarios for system development and on providing a clear semantics for them. Here, we mention only one typical paper from the area of Petri nets, where Graubmann et al. [7] propose a Petri net semantics for Message Sequence Charts. Though the use of Petri nets is not typical for this research area, this paper is typical for the following reason: The semantics of MSC's is given by a translation to an operational model (Petri nets in this case). In this way, scenarios can be considered as an operational model. Unfortunately, considering scenarios as an operational model results in all kinds of anomalies [2].

* Supported by the German DFG by grant 'Datenkonsistenzkriterien'

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 217–236, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Moreover, it spoils the way scenarios have been initially used: as incomplete fragments of executions that a system developer considers to be crucial for the system's behaviour. We claim that there is a lack of research on semantics for scenarios that faithfully reflects their incomplete character.

In this paper, we give a semantics for scenarios that reflects the incomplete character of scenarios and still allows us to argue with such scenarios. Since our approach is more general than classical scenarios such as MSCs or sequence diagrams, we call our scenarios *situations*. But, we do not give a fixed syntax for situations: We define situations semantically because we consider our work as a semantical underpinning for scenarios resp. situations rather than as a concrete notation to work with. The concrete notation may be subject to change. In our example, we will use an ad hoc notation that smoothly fits into our Petri net based approach. We have chosen Petri nets for our definition because they have a clear and simple notion of executions that reflects the causal dependence as well as the causal independence of events.

Having a precise semantics for situations allows us to specify and to verify the correctness of systems. What is more, the correctness proofs are in terms of (refined) situations and use surprisingly simple proof arguments only, which are formally expressed by *automata*. This way, the style of specification and argumentation with situations reflects the informal style in which system developers used scenarios from the very beginning.

The rest of this paper is structured as follows: In Sect. 2, we informally introduce all concepts by the help of an example: We start with a system model and its runs; then, we give examples of situations and verification techniques based on situations. In Section 3, we formalize these concepts and provide some proof rules, which formalize the informal correctness proofs from the example.

2 Example

Before the formal presentation, we give an example, which illustrates the concepts introduced in this paper and provides a flavour of their use. This example builds on previous work, where we have proposed *arc-typed Petri nets* [13] for the modelling and verification of consistency protocols. Basically, an arc-typed Petri net is an algebraic Petri net equipped with different types of arcs. The arc-types carry over to the runs of the arc-typed net and allow us to distinguish among different kinds of *causalities* in the run. In the context of consistency protocols, the distinction between the two causalities *control flow* and *data flow* plays an important role: Most consistency models can be specified in terms of read and write events, and control flow and data flow [9,10].

In [11], we have used arc-typed Petri nets for modelling and verifying a realistic protocol for the fault-tolerant execution of parallel programs. In [11], however, we concentrate on the verification of the protocol rather than on the underlying verification technique. Here, we formalize the concept of situations and provide a formal underpinning of the verification techniques. Since our focus

is on the concepts and their formalization, and not on the protocol, we use a simplified version of the protocol here.

2.1 A System Model

Figure 1 shows an arc-typed Petri net that models a simple consistency protocol. We assume that a set of sequential *processes* P is run in parallel; the sequential processes may communicate by sending and receiving messages. Moreover, there is a set O of *shared objects*, which may be read or written by the sequential processes. A *memory protocol* implements the read and write operations on the shared objects. For efficiency reasons, the memory protocol maintains copies of the same object. Clearly, the memory protocol must guarantee some kind of *consistency* for the values read by the parallel program. Actually, there are many different definitions of consistency. We require that our memory protocol should guarantee *weak consistency* [3,5], which will be defined in Sect. 2.4.

In the Petri net model, the parallel processes are represented in place proc. The possible operations of each process are modelled by the transitions W[o] (a write operation on object o), R[o] (a read operation on object o), Snd (a send operation), and Rcv (a receive operation). Note that there are two R[o] transitions: one for a read on a copy of object o and one for a read on the original of the object. The parallel program, however, will not be aware which read operation is executed; this is in the responsibility of the memory protocol. Therefore, both read operations carry the same name R[o].

A message sent to some process p' is represented by a token p' on place messages; we represent neither the contents of a message nor its sender. Note that our Petri net model does not really execute a parallel program. Rather, it executes read, write, send, and receive operations non-deterministically. This way,

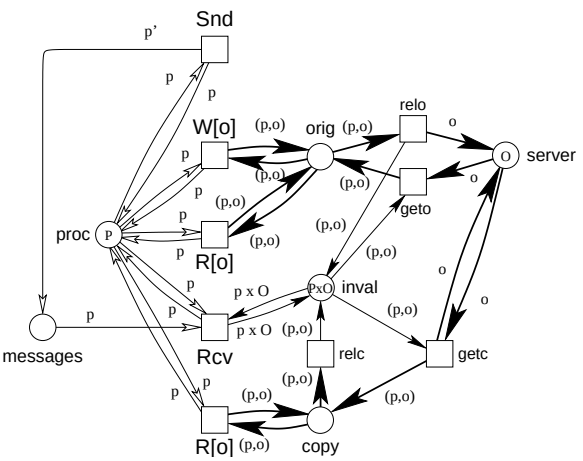


Fig. 1. A Petri net model of the memory protocol

the net simulates the execution of all parallel programs. Since the protocol must guarantee consistency for all parallel programs, this provides an appropriate abstraction.

As already mentioned, there is an original of each object and there can be several copies of each object. It is the task of the memory protocol, to keep the different copies consistent. At each time, the protocol has exactly one original for each object $o \in O$. The original resides either at a distinguished server or it resides at some process p. An original of object o residing at the server is represented by a token o on place server; an original residing at some process p is represented by a token (p, o) on place orig. A copy of some object at process p is modelled by a token (p, o) on place copy. Each process has either a copy of an object, or it has the original of the object, or the object is invalid at this process. Being invalid is represented by a token (p, o) on place inval. Initially, all originals O reside at the server, and all copies are marked as invalid for each process.

A write operation can occur only when the process has the original of the object. The read operation, however, can occur on the original of the object as well as on a copy of the object. The protocol may move the original from the server to some process (transition geto) or it may move it back to the server (transition relo). Likewise, the protocol can make a copy of the original from the server (transition getc)—in this case, the original remains on the server. Moreover, the protocol can invalidate a copy (transition relc). In order to avoid access of outdated copies, the protocol makes sure that all objects are invalid at process p, when a receive operation of p is executed. This is represented by the arcs to and from place inval, which are labelled by $p \times O$. With the above interpretation, this means that all objects O are invalid at process p, whenever p executes the receive operation.

In the Petri net model of Fig. 1, we used different arc-types. Arcs with a white arrowhead represent the *control flow* of the executed parallel program, i. e. the order in which the simulated parallel program has executed the different operations. Note that we consider a synchronization by sending a message from one process to another as control flow of the program, too, because these synchronizations are explicitly established by executing send and receive operations. The bold-faced arcs represent the propagation of data along an original or a copy of an object; it is called *data flow*. The remaining arcs do not have a specific meaning—they represent just some kind of synchronization or ordering, which is necessary for achieving consistency. We will see in Sect. 2.4 that the distinction between control flow and data flow can be used for specifying weak consistency in a nice and simple way.

2.2 Runs

Before giving this specification, let us have a look at the runs of the arc-typed Petri net. Basically, a *run* is a non-sequential process [4] of the arc-typed net, where the arc-types carry over from the arc-typed net to the non-sequential process in a canonical way. Figure 2 shows one run of the arc-typed Petri net

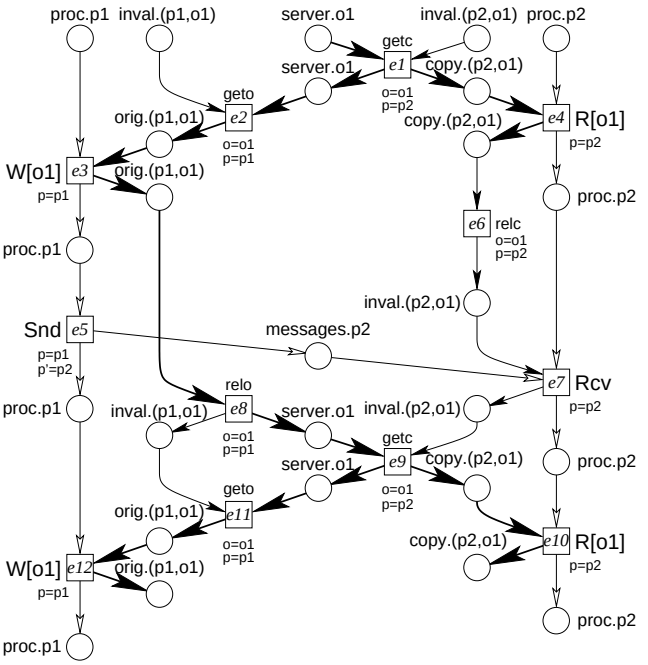


Fig. 2. A run of the arc-typed Petri net

from Fig. 1, where we assume that there is only one shared object, and that there are only two processes, i. e. $O = \{o1\}$ and $P = \{p1, p2\}$. Basically, the run is an unwinding of the arc-typed Petri net starting from the initial marking. The places of a run are called *conditions*, and its transitions are called *events*. The flow relation of a run is not branching at conditions, and the flow relation has no cycles (see [4,13] for more motivation).

Each condition of the run is labelled by a pair that consist of a place of the arc-typed Petri net and a token; we call this pair a *resource* of the arc-typed Petri net. So, each condition represents the occurrence of a resource, i. e. the occurrence of a token on a place. An event of the run corresponds to a pair that consists of a transition of the arc-typed Petri net and an assignment of values to all the involved variables of that transition (the assignment is often called a *mode*). We call such a pair an *action*. So, each event of a run represents the occurrence of an action; i. e. the occurrence of a transition in some mode. The conditions in the *context* of the event represent the resources consumed and produced by the occurrence of the action corresponding to the event.

2.3 Situations

In general, runs of an arc-typed net will be too large for drawing them and for directly arguing on them. Even worse, runs can be infinite and there can be

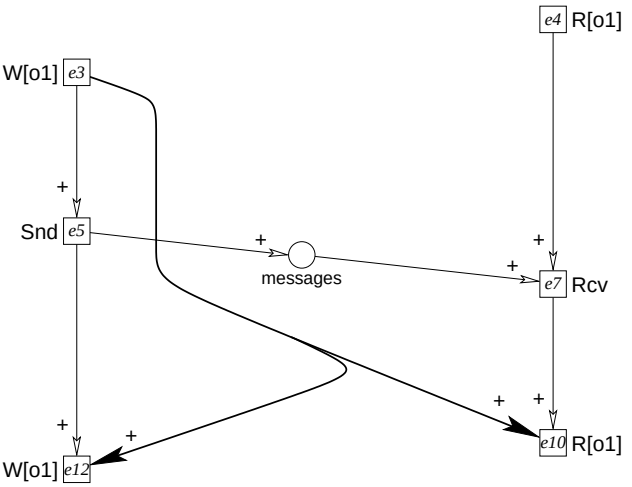


Fig. 3. A situation

infinitely many runs. Therefore, it is impossible to argue on all runs of an arc-typed Petri net directly. This is where *situations* come in. A situation captures one or more relevant aspects of one, of several, or even of all runs of the considered system. Figure 3 shows the situation of the run from Fig. 2 from a more abstract point of view. It focuses on the part that is relevant for the consistency protocol: the operations executed by the parallel program, and the control flow and the data flow between these events. Note that we have indicated the *transitive nature* of the different causalities in this situation by annotating them with +. In the example, all causalities are transitive. In some situations, however, we would like to express that there is a *direct* causality between an event and a condition. In that case, we will omit the + in order to be consistent with the notation for runs[1].

In Fig. 3, we have named the events as in the run, in order to make the correspondence of this situation to the run more explicit. In general, we can choose arbitrary names for the events (and conditions) in a situation; then, the correspondence to a particular run will be established by a mapping from the events and conditions of the situation to the events and conditions of the run. In fact, the same situation could correspond to the events and conditions of a run in several different ways.

Note that the condition in this situation is labelled by messages only. This provides another way of abstraction because the condition of the situation could be mapped to different conditions of the run: those associated with a resource messages.p1 and those associated with a resource messages.p2. Likewise, we have omitted the values assigned to the variable p for the different events of the

[1] This way, a runs can be considered as a very special situation, which denotes itself.

execution. So, we don't know any more to which process the different read and write events belong.

Altogether, a situation focuses on some events and conditions of runs, and characterizes the possible actions and resources associated with them in the run. Moreover, a situation characterizes some causal relations between these conditions and events. In some cases, we will characterize even the legal resources on a path of causality. For example, the situation on the right-hand side of Fig. 9(a) requires that there is a path of data flow arcs from event $e1$ to $e2$ such that all conditions on this path are associated with the resource $(\mathsf{copy}, (p1, o1))$. We will come back to this later.

2.4 A Specification

Next, we will give the specification of weak consistency in terms of situations. Actually, the specification is in terms of an implication between two situations. Since these implications are a concept of their own, we call them *situation implications* or *implications* for short.

For our protocol we require that each run is weakly consistent. A run is *weakly consistent* if it meets the following condition:

> Let e_1 and e_2 be two events of the run such that e_1 is a write event on some object $o1$, such that e_2 is a read or write event on the same object $o1$, and such that there is a path of control flow arcs from e_1 to e_2. Then, there is also a path of data flow arcs from e_1 to e_2.

This requirement can be formalized as a situation implication as shown in Fig. 4, where the label X[o1] represents either a read event R[o1] or a write event W[o1] on object $o1$. Note that it is crucial for the specification that, in both situations, the events with the same names refer to the same event in the corresponding execution.

In the situation implication of Fig. 4, $o1$ is a representative for any object. The implication must be valid for each concrete shared object chosen for $o1$. So actually, situations and situation implications have parameters, and there validity must be proven for each concrete value assigned to them. But, we will see that this parameterization of situations does not require extra effort in the verification. This parameterization reflects the mathematical phrase: 'Let $o1$ be an arbitrary but fixed object'.

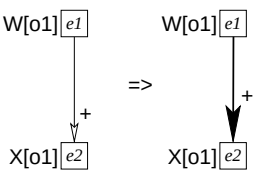


Fig. 4. Specification of weak consistency

Unfortunately, we cannot motivate the specification of weak consistency here (see [13,10]). For now, it is sufficient to know that it can be formalized as a situation implication. For verifying that the system satisfies this specification, we must show the existence of a path of data flow between two events from the existence of a path of control flow between these events.

2.5 Verification

Next, we illustrate how to prove the validity of situation implications for a system. Since our focus is on the formalization of the verification technique rather than on the verification of the protocol, we don't give a full proof of the protocol here. Rather, we present some interesting proof arguments. A full proof of a more realistic protocol can be found in [10].

First argument The first argument considers two events $e1$ and $e2$ which use the original of some object $o1$, i. e. which have an original of object $o1$ in their context. This situation is shown in the situation on the left-hand side of Fig. 5. Remember, that the original can either be an original at the server or an original at some process. This alternative is indicated in this situation by the two labels orig.(.,o1) and server.o1, where the dot in $(., o1)$ indicates that the process at which the original resides does not matter.

If $e1$ occurs causally before $e2$ as indicated by a transitive causality in the situation on the left-hand side of Fig. 5, then there is also a path of data flow arcs from $e1$ to $e2$. The informal argument is that there is always exactly one original of object $o1$, which is always propagated along data flow arcs in the Petri net model. So, there is a path of data flow arcs from condition $c1$ to $c2$. This argument can be formalized as follows: The Petri net model from Fig. 1 has a place invariant $\mathsf{server} + pr_2(\mathsf{orig}) = O$, where pr_2 projects each token on orig on its second component, which is the object. Therefore, we know that there is always exactly one token $o1$ on place server or one token $(p, o1)$ on place orig, where p is some process. Therefore, we know that there is a path from $e1$ via $c1$ and $c2$ to $e2$, on which only the resources server.o1 and orig.(.,o1) occur. Next, we investigate how this path could look like. Figure 6 shows the two places concerned along with the transitions which correspond to $e1$ and resp. $e2$ in the

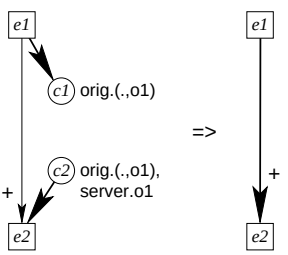


Fig. 5. The first argument

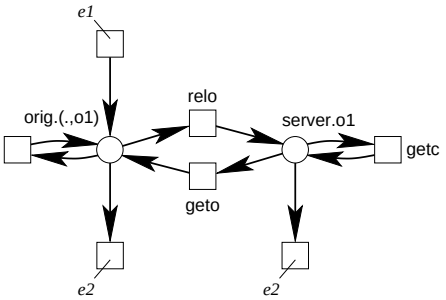


Fig. 6. An automaton representing the possible paths

situation. Note, that there are two possibilies for $e2$ because $e2$ can consume a resource orig.(.o1) or a resource server.o1. Moreover, Fig. 6 shows all possible actions of the Petri net model that consume one of the considered resources. This way, we have represented all possible paths form $e1$ to $e2$ on which only resources orig.(.,p1) and server.o1 occur. All these paths consist of data flow arcs only, which finishes the proof of the first argument.

In Figure 6, we have extracted the part of the Petri net model, which is concerned in the propagation of originals. We call it an *automaton* along an invariant. The event indicated by $e1$ is called a *start event* of the automaton, the events indicated by $e2$ are called *end events* of the automaton. A careful construction of the automaton guarantees that the path form $e1$ to $e2$ in the run along the invariant is reflected by a path form a start event to an end event of the automaton. The detailed requirements which guarantee this property will be defined in the formal part.

Note that the automaton is parameterized in the same way as situations are, since $o1$ can be chosen as any shared object. Since the concepts of parameterization of automata and of situations are basically the same, it does not bother us too much, when applying the verification techniques in practice. In the formalization of situations, of situation implications, and of automata, however, we will formally capture the parameterization.

Second argument Another important observation in the proof of the protocol is the following: Whenever there is a path of control flow from a write event $e1$ of a process $p2$ to a read event $e2$ of a process $p1$ with $p1 \neq p2$, then there is a receive event of process $p1$ on this path. This is expressed by the situation implication in Fig. 7. Informally, the argument is that a control flow between two different processes can be established only by a send operation and a receive operation. Formally, we capture all possible paths of control flow from event $e1$ to event $e2$ in an automaton again. The automaton is shown in Fig. 8. Again, the events indicated by $e1$ and $e2$ correspond to the events of the situation; $e1$ is the start event and $e2$ is the end event. The automaton has conditions, which represent all resources that may be accessed by an action with a control flow arc. In order to obtain our result, we distinguish between resources proc.p with $p \neq p1$ and

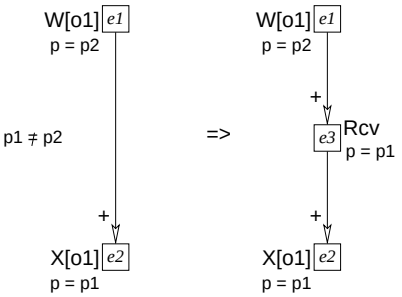


Fig. 7. The second argument

resource proc.p1. Moreover, we have a separate condition for resource messages.p (for arbitrary p). Then, we fill in all actions of the Petri net model that consume one of these resources and produce one of these resources along a control flow arc. Then, we know that the path from $e1$ to $e2$ in a run of the Petri net model is represented by a path from the start event of the automaton to the end event. Obviously, there is a Rcv event with $p = p1$ on each path of the automaton. This proofs our second argument. The detailed requirements on the construction of this automaton will be defined in the formal part.

Third argument At last, we consider a read event $e2$ of some process $p1$ that reads on a copy of some object $o1$. This situation is shown on the left-hand side of Fig. 9(a). In this situation, we know that there must be some getc event $e1$ with $o = o1$ such that there is a path of data flow from $e1$ to $e2$ on which only copy.(p1,o1)-conditions occur. This situation is shown on the right-hand side of Fig. 9(a). Informally, the argument is that the copy read by the read event must have been obtained by a getc event before. More formally, the argument is as follows: Since place copy is unmarked initially, there must have been some action that produced the resource copy.(p1,o1). The automaton shown in Fig. 9(b) traces back the events that could possibly produce this resource: It could be

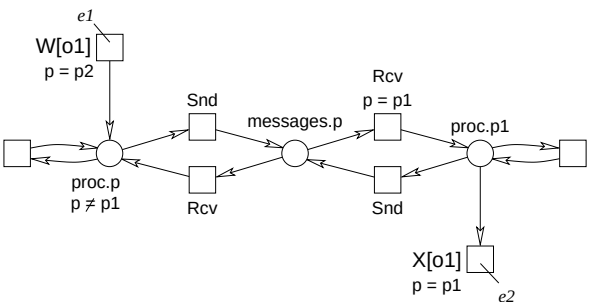
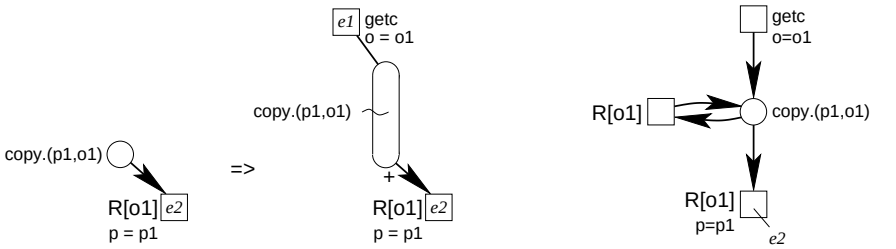


Fig. 8. An automaton representing all possible paths along control flow



(a) The third argument (b) A backwards automaton

Fig. 9. The third argument and its proof

another read event R[o1]; a read event, however, needs a resource copy.(p1,o1) itself. So, originally, there must have been some other event; the only remaining possibility is a getc event with $o = o1$. From this automaton, we know that there must be an event getc as shown on the right-hand side of the situation implication of Fig. 9(a). Again, the detailed requirements that must be met by this automaton will be defined in the formal part.

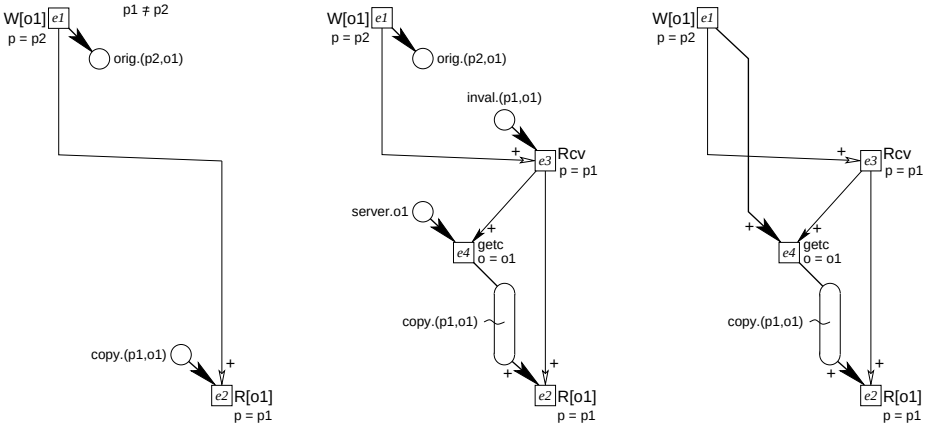
Combining the arguments Above, we have stated the three key arguments for the correctness of the memory protocol. In order to provide a flavour of how these arguments are combined to a full proof, we sketch the proof of one case[2], which is shown in Fig. 10(a): We consider a write event $e1$ and a read event $e2$ on the same object $o1$, where the events are executed by two different processes and $e2$ reads on a copy. Moreover, there is a path of control flow from $e1$ to $e2$. We must show that there is a data flow, too.

By the second argument (see Fig. 7), we know that there is a receive event between $e1$ and $e2$ as shown in Fig. 10(b). Note that we added the resource inval.(p1,o1) in the context of this event. By the third argument (see Fig. 9(a)), we know that the copy read by the read event was obtained by a getc event. By a place invariant, we know that there cannot be a copy of an object $o1$ for process $p1$ when the object is marked to be invalid. Since there is a copy of $o1$ for $p1$ between $e4$ and $e2$ throughout, we know that the Rcv event $e3$ occurs causally before $e4$. By transitivity, $e1$ occurs causally before $e4$, too. Since $e1$ and $e4$ both access the original of object $o1$, we can apply the first argument (see Fig. 5), which gives us the situation shown in Fig. 10(c): a path of data flow from $e1$ to the $e2$ (via $e4$), which finishes the proof (of the considered case).

3 Formalization

In this section, we formalize the above concepts. In the formalization, the focus is on the concepts and their semantics. Therefore, we do not give a syntactical

[2] In fact, this is the most involved case for this example.



(a) A special case (b) More details (c) The dataflow

Fig. 10. Combining the arguments

representation for arc-typed Petri nets, situations, and automata. The above examples, however, should provide some flavour of a possible graphical notation: This notation resembles the notation of algebraic nets for their semantical counterpart called *basic high-level nets* [15].

3.1 Arc-Typed Petri Nets

In this section, we formalize arc-typed Petri nets and their runs. Basically, this semantical model corresponds to basic high-level nets; it is the underlying semantical model of arc-typed Petri nets as defined in [13]. For a syntactical representation and a more detailed motivation, we refer to [13,12].

Multisets We start with some basic notation. For some set D, a mapping $m : D \to \mathbb{N}$ such that $\sum_{d \in D} m(d) \in \mathbb{N}$ is called a *bag* over D. The set of all bags over some set D is denoted by $\mathbf{B}(D)$.

Petri nets We call $N = (P, T, F)$ a *Petri net* if P and T are two disjoint sets and $F \subseteq (P \times T) \cup (T \times P)$. The elements of P are called *places*, the elements of T are called *transitions* of the net; the elements of $P \cup T$ are called the *elements of* N also. The elements of F are called *arcs*. For some element x of N, the set ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ is called the *preset of* x *in* N; the set $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$ is called the *postset of* x *in* N.

Algebraic nets In an *algebraic Petri net*, each place of the Petri net $N = (P, T, F)$ is associated with some domain, which defines the legal tokens on that place. Then, a pair (p, d) that consist of a place p and a legal token d is called a *resource* of the algebraic net. In a syntactical representation, the domain associated with

a place is represented by a sort of some signature. In our presentation, we define the set of resources $R \subseteq P \times D$ for some *domain* D semantically.

In an algebraic net, a transition can occur in different *modes*. Typically, the possible modes are syntactically represented by variables, possibly restricted by some transition guards. A pair $\langle t, \mu \rangle$ that consists of a transition t and a mode μ is called an *action*. In our presentation, we define the set of actions $A \subseteq T \times D$ semantically. For each action $a \in A$, the algebraic Petri net defines a bag of resources consumed and a bag of resources produced by an occurrence of this action. The bag of resources consumed by action a is denoted by ${}^{-}a$. The bag of resources produced by action a is denoted by a^{+}. We assume that ${}^{-}\langle t, \mu \rangle(p, d) = 0$ for $(p, t) \notin F$ and $\langle t, \mu \rangle^{+}(p, d) = 0$ for $(t, p) \notin F$; i. e. an action consumes and produces tokens along arcs of the underlying net N only. Moreover, we assume that for each action a, the bags ${}^{-}a$ and a^{+} are not empty. Technically, ${}^{-}. : A \to \mathbf{B}(R)$ and $.^{+} : A \to \mathbf{B}(R)$ are mappings that associate each action with some bag over R. In addition, each algebraic Petri net is equipped with an initial marking m_0. In our semantical presentation, the initial marking is some bag of resources: $m_0 \in \mathbf{B}(R)$.

Arc-typed nets In addition to an algebraic Petri net, an arc-typed net distinguishes among different types of arcs. This is represented by a family of sets of arc $(F_c)_{c \in C}$, where C is a finite set of *arc-types* and, for each $c \in C$, we have $F_c \subseteq F$. An arc $f \in F_c$ is called an arc of type c. For simplicity, we assume that there is a distinguished $f \in C$ with $F_f = F$.

Altogether, an *arc-typed Petri net* consists of a net $N = (P, T, F)$, some domain D, some set of resources $R \subseteq P \times D$, some set of actions $A \subseteq T \times D$, an initial marking m_0, two mappings ${}^{-}. : A \to \mathbf{B}(R)$ and $.^{+} : A \to \mathbf{B}(R)$, a set of arc-types C, and a family $(F_c)_{c \in C}$ such that the above constraints are met. We fix this arc-typed net for the rest of this paper.

Note that in the graphical representation, we have annotations such as W[o] for transitions, where o is a variable occurring in the annotations of the corresponding arcs. This annotation, however, does not have a formal meaning. It is meant to stress the fact that the write operation is on object o, which is also clear from the mode[3] in which the transition occurs..

Occurrence nets Next, we formalize the runs of an arc-typed Petri net. Basically, a run of an arc-typed Petri net is a process [4] of the Petri net as defined for algebraic Petri nets [13,12]; the different arc-types of the arc-typed net carry over to the run canonically. A run is a labelled occurrence net, where an occurrence net $K = (B, E, \lessdot)$ is a Petri net such that the transitive closure $<$ of $\lessdot$ is acyclic, such that each element of K has only finitely many predecessors with respect to $<$, and such that, for each $b \in B$, we have $|{}^{\bullet}b| \leq 1$ and $|b^{\bullet}| \leq 1$; i. e. $\lessdot$ is not branching at b (see [4] for motivation). In order to distinguish between the elements of the arc-typed net and the elements of its run, we call the places B of the occurrence net *conditions*, and we call the transitions E

[3] In this respect, arc-typed nets as proposed here slightly differ from it original version.

of the occurrence net *events*. The set of *minimal elements* of K is denoted by ${}^\circ K = \{x \in B \cup E \mid {}^\bullet x = \emptyset\}$.

Runs A run of an arc-typed net is an occurrence net along with two mappings $\rho : B \to R$ and $\rho : E \to A$. We use the same symbol ρ for both mappings because it is clear from the domain which mapping applies. For a finite subset $Q \subseteq B$ of conditions, $\rho(Q)$ denotes a bag of resources: $\rho(Q)(r) = |\{b \in Q \mid \rho(b) = r\}|$; i. e. $\rho(Q)(r)$ represents the number of conditions in Q with label r.

An occurrence net $K = (B, E, \lessdot)$ along with the two mappings ρ and a family of relations $(\lessdot_c)_{c \in C}$ with $\lessdot_c \subseteq \lessdot$ is a *run* of the arc-typed Petri net, if and only if:

1. ${}^\circ K \subseteq B$ is finite and $\rho({}^\circ K) = m_0$
2. For each $e \in E$, we have $\rho({}^\bullet e) = {}^- \rho(e)$ and $\rho(e^\bullet) = \rho(e)^+$.
3. For each pair of elements $x, y \in B \cup E$ with $x \lessdot y$ and for each $c \in C$, we have $x \lessdot_c y$ if and only if $(pr_1(\rho(x)), pr_1(\rho(y))) \in F_c$, where pr_1 denotes the projection to the first component of a pair.

The first requirement guarantees, that the minimal elements of the run correspond to the initial marking of the arc-typed Petri net. The second requirement guarantees that the occurrence of an event e in the run consumes and produces exactly those resources that are consumed and produced by the corresponding action $\rho(e)$. The third requirement guarantees that the arc-types carry over to the run. In the following, we denote a run by ρ for short.

Note that we do not require progress in our definition of runs, i. e. a run can stop at each stage. Since we are considering only verification techniques for safety properties in this paper, we do not loose anything. For considering liveness properties, we can easily add the progress assumption and fairness assumptions (cf. [1]). But this is a different issue.

Paths in a run A *path* is a sequence $\pi = x_0 c_1 x_1 c_2 x_2 c_3 \ldots c_n x_n$ such that $x_i \in A \cup R$ and $c_i \in C$, and we have: If $x_{i+1} \in A$, then $x_i \in R$; and if $x_{i+1} \in R$, then $x_i \in A$ (i. e. resources and actions alternate in the sequence of x_is). For two elements y_0 and y_n of ρ, the path π is a *path from y_0 to y_n in run ρ* if there exists a sequence $y_1, \ldots, y_{n-1}$ of elements of ρ such that $\rho(y_i) = x_i$ for each $i \in \{0, \ldots, n\}$ and $y_i \lessdot_{c_{i+1}} y_{i+1}$ for each $i \in \{0, \ldots, n-1\}$.

3.2 Situations and Implications

Now, we are prepared to define situations, situation implications and their semantics.

Situations We start with the definition of a situation. As we have seen in the informal part, a situation basically consists of a set of events E' and a set of conditions B' along with some causal relationships. A causal relationship can be either direct or transitive. In order to distinguish direct relationships from transitive relationships, we distinguish two families: $(F_c^d)_{c \in C}$ and $(F_c^t)_{c \in C}$, where C

is the set of arc-types from the arc-typed net, $F_c^d \subseteq (E' \times B') \cup (B' \times E')$, and $F_c^t \subseteq (E' \cup B') \times (E' \cup B')$ for each $c \in C$. Note that, in situations, transitive causalities between two events or between two conditions are allowed. In the graphical representation, the arcs of F_c^t are distinguished from the arcs of F_c^d by a label $+$.

Each event $e \in E'$ is associated with a set of corresponding actions, which is a subset of the actions A of the algebraic net. Each condition $b \in B'$ is associated with a set of corresponding resources, which is a subset of the resources R of the algebraic net. Remember, that situations are parameterized: In our examples, there were 'global' variables $o1$ and $p1$, which could take different values. In our semantical definition, this is reflected by the fact that the set of corresponding actions is dependent also from a value chosen from some domain. For simplicity, we chose the parameter from the domain D of the algebraic net. Then, the corresponding set of actions and resources can be formalized by two mappings: $\lambda : E' \times D \to \mathcal{P}(A)$ and $\lambda : B' \times D \to \mathcal{P}(R)$, where $\mathcal{P}(X)$ denotes the powerset of X for some set X.

As mentioned earlier, we would like to restrict the set of resources occurring on a particular path in some situations. This can be expressed analogously by a third mapping: $\lambda : F^t \times D \to \mathcal{P}(R)$, where $F^t = \bigcup_{c \in C} F_c^t$. Note, that $\lambda(f, d) = R$ does not impose any restriction, which is the default case.

Semantics of situations Next, we establish the correspondence between a situation and a run ρ. This is achieved by mapping the conditions B' and the events E' of the situation to the conditions B and the events E of the run ρ. Technically, we define two mappings $\gamma : E' \to E$ and $\gamma : B' \to B$. Moreover, we must fix a concrete value for the parameter $d \in D$ of the situation.

Given γ and d, we say that a situation σ *matches* ρ *for* γ *and* d if the following conditions are met:

- For each $e \in E'$, we have $\rho(\gamma(e)) \in \lambda(e, d)$; i. e. the action associated with $\gamma(e)$ in ρ is from the set of e's associated actions for d in the situation.
- For each $b \in B'$, we have $\rho(\gamma(b)) \in \lambda(b, d)$; i. e. the resource associated with $\gamma(b)$ in ρ is from the set of b's associated resources for d in the situation.
- For each pair $(x, y) \in F_c^d$ we have: $\gamma(x) \lessdot_c \gamma(y)$; i. e. the corresponding nodes in the run are directly related by causality c if required by the situation.
- For each pair $f = (x, y) \in F_c^t$, there exists a path π in ρ from $\gamma(x)$ to $\gamma(y)$ such that each causality of path π is in c, and each resource of path π is in $\lambda(f, d)$.

Situation Implication A *situation implication* is a pair of two situations (σ, σ'), where the set of conditions and events of σ' is a superset[4] of the conditions and events of σ. We denote this pair also by $\sigma \Rightarrow \sigma'$. A run ρ *meets the situation implication* $\sigma \Rightarrow \sigma'$ if for each $d \in D$ and each γ such that σ matches ρ for γ

[4] Note that, in a graphical representation, it is often convenient to omit some conditions or events on the right-hand side of an implication. But, these events can easily be added in the formal representation. So, this is not a serious restriction.

and d, there exists an extension γ' of γ such that σ' matches ρ for γ' and d. Note that this definition means, that we cannot change the parameter d and the conditions and events associated with the situation σ, when looking for a matching map γ' for σ'.

An arc-typed net meets a situation implication if each run of the arc-typed net meets the situation implication.

3.3 Forward Automata

Clearly, the paths in a run of an arc-typed Petri are represented in the arc-typed net itself. But, most interesting paths are not reflected in the graphical structure of the net, but are somehow hidden below the surface. In this section we, introduce *automata*, which explicitly reflect the paths between two events of a run in the graphical structure of the automaton. We introduce automata in two steps. First, we define the general structure of an automaton as a particular Petri net. Then, we define additional constraints that guarantee that the automaton represents at least one path between two events in the considered situation.

If the automaton represents at least one path, we can easily infer paths for the considered situation from that automaton. This way, we can proof that an arc-typed net meets a situation implication (see Sect. 2.5 for examples). So it remains to guarantee that, in a particular situation, the automaton represents at least one path between these events.

Automata An *automaton* is a special Petri net equipped with some labels. For simplicity, we use the same symbols for the underlying Petri net of an automaton as we used for occurrence nets: $K = (B, E, \lessdot)$. For an automaton, however, we do not require $\lessdot$ to be acyclic, and we do not require that conditions are non-branching. Rather, we require that events are non-branching: for each event $e \in E$ we require $|{}^\bullet e| \leq 1$ and $|e^\bullet| \leq 1$. An event $e \in E$ with ${}^\bullet e = \emptyset$ is called a *start event*; an event with $e^\bullet = \emptyset$ is called an *end event* of the automaton.

As in situations, each event of an automaton is associated with a subset of actions of the algebraic net; each condition of an automaton is associated with a subset of resources. Likewise, the associated set can depend on some parameter $d \in D$. These associated sets are represented by two mappings $\alpha : E \times D \to \mathcal{P}(A)$ and $\alpha : B \times D \to \mathcal{P}(R)$. Moreover, an automaton is equipped with a family of arcs $(\lessdot_c)_{c \in C}$, such that $\lessdot_c \subseteq \lessdot$ for each $c \in C$, and such that $\lessdot = \bigcup_{c \in C} \lessdot_c$; i. e. each arc $x \lessdot y$ is represented by some arc $x \lessdot_c y$ for some $c \in C$. Since we introduced a distinguished $f \in C$ that reflects all arcs of the arc-typed net, it is always possible to meet this condition.

A path $\pi = a_0 c_1 r_1 c_1' a_1 c_2 r_2 c_2' a_2 \ldots a_n$ is a path of the automaton for $d \in D$ if there is a sequence of events $e_0, \ldots, e_n \in E$ of the automaton and a sequence of conditions $b_1, \ldots, b_n \in B$ of the automaton such that e_0 is a start event and e_n is an end event, such that $a_i \in \alpha(e_i, d)$ for each $i \in \{0, 1, \ldots, n\}$, such that $r_i \in \alpha(b_i, d)$ for each $i \in \{1, \ldots, n\}$, and such that $e_{i-1} \lessdot_{c_i} b_i$ and $b_i \lessdot_{c_i'} e_i$ for each $i \in \{1, \ldots, n\}$.

Correspondence Next, we impose additional requirements on automata. These requirements guarantee that an automaton contains at least one possible path between two different events e' and e'' of a run ρ of the arc-typed Petri net. We consider two sets of actions A_1 and A_2, with $\rho(e') \in A_1$ and $\rho(e'') \in A_2$, and we assume that there is a path from e' to e'' in the run.

If we have an automaton and a $d \in D$ such that the following four requirements are met, then there exists at least one path of the automaton for d that is also a path from e' to e'' in the run ρ:

1. For each action $a \in A_1$ and each resource $r \in R$ of the arc-typed net with $a^+(r) \geq 1$, there exists a start event $e \in E$ and a condition $b \in B$ of the automaton such that $a \in \alpha(e, d)$, $r \in \alpha(b, d)$, and $e \lessdot b$.
2. For each condition $b \in B$ of the automaton and each resource $r \in \alpha(b, d)$ holds: For each action $a \in A$ with ${}^-a(r) \geq 1$ and each resource r' with $a^+(r') \geq 1$, there exists an event $e \in E$ and a condition $b' \in B$ of the automaton such that $a \in \alpha(e, d)$, such that $r' \in \alpha(b', d)$, such that $b \lessdot e$, and such that $e \lessdot b'$.
3. For each condition $b \in B$ of the automaton and each resource $r \in \alpha(b, d)$ holds: For each action $a \in A_2$ with ${}^-a(r) \geq 1$, there exists an end event $e \in E$ of the automaton with $a \in \alpha(e, d)$ and $b \lessdot e$.
4. For each two elements $x, y \in B \cup E$ of the automaton and each $c \in C$ with $x \lessdot_c y$ holds: For each $x' \in \alpha(x, d)$ and each $y' \in \alpha(y, d)$ with $(pr_1(x'), pr_1(y')) \in F$ we have also $(pr_1(x'), pr_1(y')) \in F_c$.

This can be proven by induction on the path from e' to e'' in ρ—exploiting the definition of runs (in particular requirements 2 and 3) and the above requirements on the automaton. An automaton that meets the above requirements is called a *forward automaton* (with respect to the arc-typed net and A_1 and A_2).

The above requirements on the automaton may appear a little awkward. For example, requirements 1 and 2 would be much simpler if we considered one arc at a time. Rather, we consider the existence of an arc from a condition to some event and from the event to some condition at a time (cf. requirement 2). The reason for our more complicated requirements is that this definition provides simpler automata. Since simpler automata are easier to construct and more comprehensible, the application of the proof techniques and the proofs themselves become simpler with our more complicated definition. For example, it would be necessary to precisely define the set of actions corresponding to the event on the right-hand side of the automaton in Fig. 8. With our more complicated definition, we need not characterize the associated actions of this event at all; i. e. the event is associated with the set of all actions A.

Special cases In some cases, we know by some other sources that the path from e' to e'' has some specific properties. For example, it could be that the path from e' to e'' is along a specific causality $c \in C$, which is expressed on the left-hand side of some situation implication. In that case, it is sufficient to consider only arcs from F_c in the above requirements—the other arcs can be safely ignored. We call this an *automaton along causality c*. Typically, this simplifies the automaton

and allows us to characterize a path between two events e' and e'' in a run more precisely. In Fig. 8, we have seen an automaton along control flow already. This automaton characterizes all paths of control flow between a write event of process $p2$ and some access event of a different process $p1$. In particular, it shows that there must be a receive event of process $p1$ in between. In this automaton, the set of resources associated with the left condition is $\{(\mathsf{proc}, p) \mid p \in P, p \neq p1\}$. Syntactically, this is represented by a variable p in the label $\mathsf{proc.p}$. The range of this variable, however, is restricted by $p \neq p1$. Similarly, the middle condition is associated with the set of resources $\{(\mathsf{messages}, p) \mid p \in P\}$; note that the range of p is not restricted in this condition.

Another special case of an automaton, is a path along some specific resources $R' \subseteq R$; the existence of such a path could, for example, be known from some invariant. In that case, it is sufficient to consider only resources from R' in the above requirements. We call this an *automaton along* R' or—if R' follows from some invariant—an *automaton along an invariant*. In Fig. 6, we have seen an automaton along an invariant already. In this automaton, the label $\mathsf{orig.(.,o1)}$ represents the set of resources $\{(\mathsf{orig}, (p, o1)) \mid p \in P\}$, i. e. the dot in $(., o1)$ is a don't care value.

3.4 Backward Automata

Sometimes, we want to investigate the causes of a particular event e'' in a run ρ. We use *backward automata* for this purpose. A backward automaton is also an automaton, but the correspondence requirements are formulated backwards, starting from the end event. Let us assume that $\rho(e'') \in A_2$ for some set of actions $A_2 \subseteq A$ and let $d \in D$. An automaton is a *backward automaton* with respect to A_2 if the following requirements are met:

1. For each action $a \in A_2$, there exists an end event $e \in E$ of the automaton with $a \in \alpha(e, d)$, and there exists some resource $r \in R$ and some condition $b \in B$ of the automaton such that ${}^{-}a(r) \geq 1$, $m_0(r) = 0$, $r \in \alpha(b, d)$, and $b \lessdot e$.
2. For each condition $b \in B$ of the automaton, each resource $r \in \alpha(b, d)$, and each action $a \in A$ with $a^{+}(r) \geq 1$, there exists an event $e \in E$ of the automaton with $a \in \alpha(e, d)$ and $e \lessdot b$ such that either
   (a) e is a start event of the automaton, or
   (b) there exists a resource $r' \in R$ with ${}^{-}a(r') \geq 1$, $m_0(r') = 0$ and there exists a condition $b' \in B$ of the automaton with $r' \in \alpha(b', d)$ and $b' \lessdot e$.
3. For each two elements $x, y \in B \cup E$ of the automaton and each $c \in C$ with $x \lessdot_c y$ holds: For each $x' \in \alpha(x, d)$ and each $y' \in \alpha(y, d)$ with $(pr_1(x'), pr_1(y')) \in F$ we have also $(pr_1(x'), pr_1(y')) \in F_c$.

If we have a backward automaton with respect to A_2, and if we have an event e'' of some run with $\rho(e'') \in A_2$, then there exists some path π of the automaton for d such that there exists an event e' of the run and π is a path from e' to e'' in the run.

Again, the proof is by induction on a path in the run (backwards from the event e''). The proof exploits that a condition b of a run cannot be a minimal element of the run if $m_0(\rho(b)) = 0$—thus there is an event $e \in {}^\bullet b$ in the run. Figure 9(b) shows an example of a backward automaton.

Note that, in the case of a backward automaton, we can infer not only the existence of some path between two events; we can infer the existence of some start event in the considered situation too.

4 Conclusion

In this paper, we have presented a semantical underpinning of a proof technique based on arc-typed Petri nets, situations, and automata. In particular, we have given a semantics for situations, which are abstract representations of parts of runs. Here, we could only sketch the usability of the resulting proof techniques; complete proofs (with a slightly extended notation for situations) can be found in [11,10].

Up to now, these techniques have been applied to the verification of consistency protocols and related protocols. The definitions of situations and situation implications, however, nicely reflect the informal use of scenarios when developing distributed systems and can be considered as a generalization of classical scenarios such as MSCs or sequence diagrams. It is easy to express standard MSCs in terms of situations; we can even express some liveness requirements by situation implications, which is an important issue for MSCs [6]. Moreover, the verification techniques proposed here, could be used as a formal foundation for informal arguments on MSCs or sequence diagrams. An appropriate notation, necessary conceptual and semantical extensions, and verification techniques tuned to this application field is subject to future research.

Up to now, the verification techniques have been applied by hand; the semantical conditions formalized in this paper have been checked by careful inspection. However, experience with the automatic checking of similar verification conditions in DAWN-proofs [1] suggests that the technique proposed here can be automated by the help of automated theorem provers. This is subject to future research.

References

1. Thomas Baar, Ekkart Kindler, and Hagen Völzer. Verifying intuition – ILF checks DAWN proofs. In S. Donatelli and J. Kleijn, editors, *Application and Theory of Petri Nets 1999, 20th International Conference*, *LNCS* 1639, pages 404–423. Springer, June 1999. 230, 235
2. Hanêne Ben Abdallah and Stefan Leue. Syntactical analysis of Message Sequence Chart specifications. Technical Report 96-12, University of Waterloo, November 1996. 217

3. John K. Bennet, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176. ACM, March 1990. 219
4. Eike Best and César Fernández. *Nonsequential Processes*, *EATCS Monographs on Theoretical Computer Science* 13. Springer-Verlag, 1988. 220, 221, 229
5. Lothar Borrmann and Martin Herdieckershoff. A coherency model for virtually shared memory. In *International Conference on Parallel Processing*, August 1990. 219
6. Werner Damm and David Harel. LCS's: Breathing life into message sequence charts. In *Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'99, IFIP TC6/WG6.1, 1999. 235
7. Peter Graubmann, Ekkart Rudolph, and Jens Grabowski. Towards a Petri net based semantics definition for message sequence charts. In O. Færgemand and A Sarma, editors, *SDL '93 Using Objects, proceedings of the Sixth SDL Forum*, pages 415–418. North-Holland, October 1993. 217
8. ITU-T Recommendation Z.120. Message sequence charts (MSC). ITU, 1993/96. 217
9. Ekkart Kindler. A classification of consistency models. Technical Report B99-14, Freie Universität Berlin, Institut für Informatik, October 1999. 218
10. Ekkart Kindler. *Systematische Spezifikation und Verifikation von Datenkonsistenzprotokollen*. Habilitation thesis, Humboldt-Universität zu Berlin, Institut für Informatik, August 2000. 218, 224, 235
11. Ekkart Kindler and Dennis Shasha. Verifying a design pattern for the fault-tolerant execution of parallel programs. Technical Report TR2000-803, New York University, Courant Institute of Mathematical Sciences, Computer Science Department, June 2000. 218, 235
12. Ekkart Kindler and Hagen Völzer. Flexibility in algebraic nets. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998, 19^{th} International Conference*, *LNCS* 1420, pages 345–364. Springer-Verlag, June 1998. 228, 229
13. Ekkart Kindler and Rolf Walter. Arc-typed Petri nets. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996*, *LNCS* 1091, pages 289–306. Springer-Verlag, June 1996. 218, 221, 224, 228, 229
14. James Rumbaugh, Ivar Jacobsen, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley, 1999. 217
15. Einar Smith. Principles of high-level net theory. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, *LNCS* 1491, pages 174–210. Springer, 1998. 228

Reproducibility of the Empty Marking

Kurt Lautenbach

University of Koblenz-Landau
Universitätsstr. 1, 56075 Koblenz, Germany
laut@uni-koblenz.de
http://www.uni-koblenz.de/~laut

Abstract. The main theorem of the paper states that the empty marking $\emptyset$ is reproducible in a p/t-net $\mathcal{N}$ if and only if there are reproducing T-invariants whose net representations have neither traps nor co-traps (deadlocks, siphons). This result is to be seen in connection to modeling processes, since all processes which have a start and a goal event usually reproduce the empty marking.

1 Introduction

This paper deals with a necessary and sufficient condition for the reproducibility of the empty marking in place/transition nets. First of all, this is a theoretical result. It will be shown that there are interesting connections to other notions, for example to the forward and backward liveness of the empty marking. On the other hand, there are application oriented aspects of that result. All processes with a start and a goal event usually reproduce the empty marking. Examples can be found in flexible manufacturing, workflow modeling, automatic control, ets. Nevertheless, to our knowledge there are no specific results dealing with the reproducibility of the empty marking.

The paper is organized as follows. In the second section one finds the net theoretic preliminaries. The main theorem of the third section provides a necessary and sufficient condition for the reproducibility of the empty marking $\emptyset$.

I am greatly indebted to Jörg Müller, Lutz Priese, Carlo Simon, Einar Smith and Paula Swatman for valuable comments and for pushing me to publish the material.

2 Net Theoretical Preliminaries

In this section some fundamentals of net theory will be (very) briefly introduced.

2.1 Place/Transition Nets

Definition 1. *1. A* place/transition net (p/t-net) *is a quadruple* $\mathcal{N} = (S, T, F, W)$ *where*

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 237–253, 2002.
© Springer-Verlag Berlin Heidelberg 2002

(a) *S and T are finite, non empty, and disjoint sets. S is the set of* places *(in the figures represented by cycles). T is the set of* transitions *(in the figures represented by boxes).*
(b) *$F \subseteq (S \times T) \cup (T \times S)$ is the set of directed* arcs.
(c) *$W : F \to \mathbb{N}\backslash\{0\}$ assigns a* weight *to every arc.*
In case of $W : F \to \{1\}$, we will write $\mathcal{N} = (S, T, F)$ as an abridgement.

2. *The preset (postset) of a node $x \in S \cup T$ is defined as ${}^\bullet x = \{y \in S \cup T | (y, x) \in F\}$ ($x^\bullet = \{y \in S \cup T | (x, y) \in F\}$).*
The preset (postset) of a set $H \subseteq S \cup T$ is ${}^\bullet H = \bigcup_{x \in H} {}^\bullet x$ $\left(H^\bullet = \bigcup_{x \in H} x^\bullet\right)$.

For all $x \in S \cup T$ it is assumed that $|{}^\bullet x| + |x^\bullet| \geqq 1$ holds; i.e. there are no isolated nodes.
3. *A place p (transition t) is* shared *iff $|{}^\bullet p| \geqq 2$ or $|p^\bullet| \geqq 2$ ($|{}^\bullet t| \geqq 2$ or $|t^\bullet| \geqq 2$).*
4. *A place p is an* input (output) boundary place *iff ${}^\bullet p = \emptyset$ ($p^\bullet = \emptyset$).*
5. *A transition t is an* input (output) boundary transition *iff ${}^\bullet t = \emptyset$ ($t^\bullet = \emptyset$).*

Definition 2. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net.*

1. *A* marking *of $\mathcal{N}$ is a mapping $M : S \longrightarrow \mathbb{N}$. $M(p)$ indicates the number of* tokens *on p under M. $p \in S$ is* marked by *M iff $M(p) \geqq 1$. $H \subseteq S$ is* marked by *M iff at least one place $p \in H$ is marked by M. Otherwise p and H are* unmarked*, respectively.*
2. *A transition $t \in T$ is* enabled *by M, in symbols $M[t\rangle$, iff*

$$\forall p \in {}^\bullet t : M(p) \geqq W((p, t)).$$

3. *If $M[t\rangle$, the transition t may* fire *or* occur*, thus leading to a new marking M', in symbols $M[t\rangle M'$, with*

$$M'(p) := \begin{cases} M(p) - W((p,t)) & \text{if } p \in {}^\bullet t \backslash t^\bullet \\ M(p) + W((t,p)) & \text{if } p \in t^\bullet \backslash {}^\bullet t \\ M(p) - W((p,t)) \ + \ W((t,p)) & \text{if } p \in {}^\bullet t \cap t^\bullet \\ M(p) & \text{otherwise} \end{cases}$$

for all $p \in S$.
4. *The set of all markings* reachable *from a marking M_0, in symbols $[M_0\rangle$, is the smallest set such that*

$$M_0 \in [M_0\rangle$$
$$M \in [M_0\rangle \ \wedge \ M[t\rangle M' \Longrightarrow M' \in [M_0\rangle.$$

$[M_0\rangle$ is also called the set of follower markings *of M_0.*
5. *$\sigma = t_1 \ldots t_n$ is a* firing sequence *or* occurrence sequence *for transitions $t_1, \ldots, t_n \in T$ iff there exist markings $M_0, M_1, \ldots, M_n$ such that*

$$M_0[t_1\rangle M_1[t_2\rangle \ \ldots \ [t_n\rangle M_n \text{holds;}$$

in short $M_0[\sigma\rangle M_n$. $M_0[\sigma\rangle$ denotes that σ starts from *M_0. The* firing count *$\overline{\sigma}(t)$ of t in σ indicates how often t occurs in σ. The (column) vector of firing counts is denoted by $\overline{\sigma}$.*

6. *The pair* $(\mathcal{N}, M_0)$ *for some marking* M_0 *of* $\mathcal{N}$ *is a* p/t-system *or a* marked p/t-net. M_0 *is the* initial marking.
7. *A marking* $M \in [M_0\rangle$ *is* reproducible *iff there exists a marking* $M' \in [M\rangle$, $M' \neq M$ s.t. $M \in [M'\rangle$.

Definition 3. *Let* $\mathcal{N} = (S, T, F, W)$ *be a* p/t*-net and* M_0 *a marking of* $\mathcal{N}$.

1. *A transition* $t \in T$ *is* live under M_0 *or in* $(\mathcal{N}, M_0)$ *iff* $\forall M \in [M_0\rangle\ \exists M' \in [M\rangle : M'[t\rangle$.
2. *A transition* t *is* dead *in* $(\mathcal{N}, M_0)$ *iff* $\forall M \in [M_0\rangle : t$ *is not enabled.*
   $(\mathcal{N}, M_0)$ *or* M_0 *is* dead *iff* $\nexists t \in T : M_0[t\rangle$.
3. $(\mathcal{N}, M_0)$ *or* M_0 *is* weakly live *(deadlock-free) iff* $\forall M \in [M_0\rangle\ \exists t \in T : M[t\rangle$.
4. $(\mathcal{N}, M_0)$ *or* M_0 *is* live *iff* $\forall t \in T$: t *is live under* M_0.
5. *A place* $p \in S$ *is* bounded under M_0 *iff* $\exists k \in \mathbb{N}\ \forall M \in [M_0\rangle : M(p) \leqq k$.
   $(\mathcal{N}, M_0)$ *or* M_0 *is* bounded *iff* $\forall p \in S : p$ *is bounded under* M_0.
6. *A place* p *is* markable *in* $(\mathcal{N}, M_0)$ *iff* $\exists M \in [M_0\rangle : M(p) > 0$.
   A set $A \subseteq S$ *is* markable *in* $(\mathcal{N}, M_0)$ *iff* $\exists p \in A : p$ *is markable in* $(\mathcal{N}, M_0)$.

2.2 Place Vectors and Transition Vectors

Definition 4. *Let* $\mathcal{N} = (S, T, F, W)$ *be a* p/t*-net;*

1. $\mathcal{N}$ *is* pure *iff* $\nexists (x, y) \in (S \times T) \cup (T \times S) : (x, y) \in F \wedge (y, x) \in F$.
2. *A* place vector (*$|S|$-vector*) *is a column vector* $v : S \longrightarrow \mathbb{Z}$ *indexed by* S.
3. *A* transition vector (*$|T|$-vector*) *is a column vector* $\omega : T \longrightarrow \mathbb{Z}$ *indexed by* T.
4. *The* incidence matrix *of* $\mathcal{N}$ *is a matrix* $[\mathcal{N}] : S \times T \longrightarrow \mathbb{Z}$ *indexed by* S *and* T *such that*

$$[\mathcal{N}](p,t) = \begin{cases} -W((p,t)) & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ W((t,p)) & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ -W((p,t)) + W((t,p)) & \text{if } p \in {}^\bullet t \cap t^\bullet \\ 0 & \text{otherwise} \end{cases}$$

Column vectors of which all entries equal 0 (1) are denoted by $\mathbf{0}$ $\mathbf{(1)}$. v^t *and* A^t *are the* transposes *of a vector* v *and a matrix* A*, respectively. The columns of* $[\mathcal{N}]$ *are* $|S|$*-vectors, the rows of* $[\mathcal{N}]$ *are transposes of* $|T|$*-vectors. Markings are representable as* $|S|$*-vectors, firing count vectors as* $|T|$*-vectors. The* $|S|$*-vector* $\mathbf{0}$ *denotes the empty marking* $\emptyset$.

Remark 1. In the following, we assume that all p/t-nets are pure. The reason for that is that we want a one-to-one correspondence between p/t-nets and their incidence matrices. This is no major restriction since

without altering relevant behaviour characteristics.

Definition 5. *Let i be a place vector and j a transition vector of $\mathcal{N} = (S, T, F, W)$.*

1. *i is a* place invariant (p-invariant) *iff* $i \neq \mathbf{0}$ *and* $i^t \cdot [N] = \mathbf{0}^t$
2. *j is a* transition invariant (t-invariant) *iff* $j \neq \mathbf{0}$ *and* $[N] \cdot j = \mathbf{0}$
3. $||i|| = \{p \in S \mid i(p) \neq 0\}$ *and* $||j|| = \{t \in T \mid j(t) \neq 0\}$ *are the* supports *of i and j, respectively.*
4. *A p-invariant i (t-invariant j) is*
   - non-negative *iff* $\forall p \in S : i(p) \geqq 0$ $(\forall t \in T : j(t) \geqq 0)$
   - positive *iff* $\forall p \in S : i(p) > 0$ $(\forall t \in T : j(t) > 0)$
   - minimal *iff* *i (j) is non-negative*

     and $\nexists$ *p -invariant* $i' : ||i'|| \subsetneqq ||i||$ ($\nexists$ *t -invariant* $j' : ||j'|| \subsetneqq ||j||$)
     and the greatest common divisor of all entries of $i(j)$ is 1
5. *The* net representation $\mathcal{N}_i = (S_i, T_i, F_i, W_i)$ of a p-invariant i *is defined by*

$$\begin{aligned} S_i &:= ||i|| \\ T_i &:= {}^\bullet S_i \cup S_i^\bullet \\ F_i &:= F \cap ((S_i \times T_i) \cup (T_i \times S_i)) \\ W_i &\text{ is } \textit{the restriction of } W \textit{ to } F_i. \end{aligned}$$

   The net representation $\mathcal{N}_j = (S_j, T_j, F_j, W_j)$ of a t-invariant j *is defined by*

$$\begin{aligned} T_j &:= ||j|| \\ S_j &:= {}^\bullet T_j \cup T_j^\bullet \\ F_j &:= F \cap ((S_j \times T_j) \cup (T_j \times S_j)) \\ W_j &\text{ is the restriction of } W \textit{ to } F_j. \end{aligned}$$

6. *$\mathcal{N}$ is* covered *by a p-invariant i (t-invariant j) iff* $\forall p \in S : i(p) \neq 0$ $(\forall t \in T : j(t) \neq 0)$

Proposition 1. *Let $(\mathcal{N}, M_0)$ be a p/t-system, i a p-invariant; then*

$$\forall M \in [M_0\rangle : i^t \cdot M = i^t \cdot M_0.$$

Proposition 2. *Let $(\mathcal{N}, M_0)$ be a p/t-system, $M_1 \in [M_0\rangle$ a follower marking of M_0, and σ a firing sequence that reproduces $M_1 : M_1[\sigma\rangle M_1$; then the firing count vector $\overline{\sigma}$ of σ is a t-invariant.*

Definition 6. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net, M_0 a marking of N, and $r \geqq \mathbf{0}$ a $|T|$-vector; r is* realizable in $(\mathcal{N}, M_0)$ *iff there exists a firing sequence σ with $M_0[\sigma\rangle$ and $\overline{\sigma} = r$.*

Proposition 3. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net, M_1 and M_2 markings of $\mathcal{N}$, and σ a firing sequence s.t. $M_1[\sigma\rangle M_2$; then the linear relation*

$$M_1 + [\mathcal{N}]\overline{\sigma} = M_2 \quad \text{holds.}$$

In the above linear relation, the *state equation*, the order of transition firings is lost.

Let H be a set of places (transitions); the $|S|$-vector ($|T|$-vector) $\chi(H)$ with

$$\chi(H)(h) = \begin{cases} 1 \text{ if } h \in H \\ 0 \text{ if } h \notin H \end{cases}$$

is the *characteristic vector* of H.

A set of places H is *markable* in $(\mathcal{N}, M_0)$ iff there is a marking $M \in [M_0\rangle$ s.t. $\chi(H)^t \cdot M > 0$.

Next, we will introduce the reachability graph or marking graph. Because of their size, reachability graphs are unsuitable for practical problems. As a conceptual means, they are indispensable.

Definition 7. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net and M_0 a marking of $\mathcal{N}$;*

the reachability graph $RG(\mathcal{N}, M_0) = (V, E)$ *of $(\mathcal{N}, M_0)$ is a directed graph with*

$$V = [M_0\rangle \text{ as set of nodes and}$$
$$E = \left\{(M, t, M') \mid M \in [M_0\rangle \wedge t \in T \wedge M[t\rangle M'\right\}$$
as set of labeled arcs.

$$L(E) = \left\{t \in T \mid \exists\,(M, t, M') \in E\right\} \text{ is the set of labels.}$$

2.3 Traps, Co-traps and Liveness

Definition 8. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net;*

1. *A non-empty set $H \subseteq S$ of places is a* trap *iff* $H^\bullet \subseteq {}^\bullet H$.
2. *A non-empty set $H \subseteq S$ of places is a* co-trap *(or a* deadlock, siphon*) iff* ${}^\bullet H \subseteq H^\bullet$.

3. *A trap (co-trap) h is* minimal *iff there is no trap (co-trap) contained in H as a proper subset.*

Once a trap is marked it remains marked. Once a co-trap is unmarked it remains unmarked. The supports $||i||$ of all non-negative p-invariants i are traps and co-traps.

Co-traps play an outstanding role for the weak liveness of marked p/t-nets.

This is demonstrated by the following two lemmata.

Definition 9. *Let $(\mathcal{N}, M)$ be a p/t-system with $\mathcal{N} = (S, T, F, W)$; let $p \in S$ and $D \subseteq S$;*

p is t-undermarked by M *for some $t \in T$ iff $M(p) < W(p, t)$;*

p is undermarked by M *iff there exists a $t \in T$ such that p is t-undermarked by M.*

D is undermarked by M *iff all $p \in D$ are undermarked by M.*

Lemma 1. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net and M a marking of $\mathcal{N}$;*

if $(\mathcal{N}, M)$ is dead the set of all places that are undermarked by M is a co-trap.

Proof. Let $(\mathcal{N}, M)$ be dead. If the set $H := \{p \in S \mid p$ undermarked by $M\}$ is no co-trap, then there is a transition $u \in {}^\bullet H$ s.t. $u \notin H^\bullet$. This implies ${}^\bullet u \cap H = \emptyset$ saying that all input places of u are not undermarked by M. So, all places of ${}^\bullet u$ are for no $t \in T$-undermarked by M, in particular not for u itself. Consequently, u is enabled by M in contrast to the fact that $(\mathcal{N}, M)$ is dead.

Lemma 2. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net and M_0 a marking of $\mathcal{N}$;*

if no minimal co-trap of $\mathcal{N}$ can get undermarked, $(\mathcal{N}, M_0)$ is weakly live.

Proof. Suppose, $(\mathcal{N}, M_0)$ is not weakly live. Then there is a marking $M \in [M_0\rangle$ s.t. $(\mathcal{N}, M)$ is dead. Then $H := \{p \in S \mid p$ is undermarked by $M\}$ is a co-trap. Consequently the minimal sub-co-traps of H are undermarked by M in contrast to the assumption. So $(\mathcal{N}, M_0)$ is weakly live.

Clearly, in nets whose arc weights are equal to 1, $p \in S$ and $D \subseteq S$ are undermarked by a marking M iff $M(p) = 0$ and $M(s) = 0$ for all $s \in D$ hold.

2.4 Synchronization Graphs

Definition 10. *A p/t-net $\mathcal{N} = (S, T, F)$ is a* synchronization graph (marked graph) *iff for all $p \in S$*

$$|\,{}^\bullet p\,| = |\,p^\bullet\,| = 1 \quad \text{holds.}$$

For synchronization graphs the following well known statements hold.

Lemma 3. *Let $G = (S, T, F)$ be a synchronization graph and M_0 a marking of G;*

1. *$H \subseteq S$ is a trap or a co-trap iff H is the place set of a cycle.*
2. *(G, M_0) is live iff all cycles are marked by M_0.*

3 Reproduction

Subject of this section is the following reproduction theorem and its proof. The theorem provides a necessary and sufficient condition for the reproducibility of the empty marking in a p/t-net $\mathcal{N} = (S, T, F)$.

Theorem 1 (Reproduction Theorem). *Let $\mathcal{N} = (S, T, F)$ be a p/t-net and r a non-negative t-invariant of $\mathcal{N}$;*

then in $\mathcal{N}$ the empty marking $\emptyset$ is reproducible by realizing $k \cdot r$ for $k \in \mathbb{N} \setminus \{0\}$ iff the net representation $\mathcal{N}_r$ of r does neither contain a trap nor a co-trap.

We start proving by showing that the *reproduction condition*, i.e. the non-existence of a trap or a co-trap in $\mathcal{N}_r$, is necessary. For that we need the following lemma 4 and theorem 2 whose corollary 3 states the necessity.

Lemma 4. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net and $\emptyset$ (the empty marking) its initial marking;*

then every transition that is firable at all in $(\mathcal{N}, \emptyset)$ is firable arbitrarily often in $(\mathcal{N}, \emptyset)$.

Proof. Let $t \in T$ be a firable transition of $(\mathcal{N}, \emptyset)$. Then a firing sequence σ exists with t as last transition and such that $\emptyset[\sigma\rangle$ holds. Then $M[\sigma\rangle$ holds for every other marking M of $\mathcal{N}$ such that σ can be repeated arbitrarily often.

Theorem 2. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net;*

$(\mathcal{N}, \emptyset)$ is live iff $\mathcal{N}$ contains no co-trap.

Proof. Let $(\mathcal{N}, \emptyset)$ be live. Then all transitions are firable. So, there are no places $p \in S$ with ${}^\bullet p = \emptyset$ (input boundary places). Consequently and since there exist no isolated nodes, every place is output place of some transition and can thus be marked. This, however, is impossible if $\mathcal{N}$ contains a co-trap D. In that case D is empty under $\emptyset$ and none of its places can be marked.

Assume now that $\mathcal{N}$ contains no co-trap. Then there are no input boundary places because every such place is a minimal co-trap. Let U be the set of unmarkable places of $(\mathcal{N}, \emptyset)$ and let $U \neq \emptyset$. Then ${}^\bullet U \neq \emptyset$ since U contains no input boundary places. Let be $t \in {}^\bullet U$. Then t is not firable. If all input places of t can be marked at all, they can be marked sufficiently high according to lemma 4. So, the reason for the non-firability of t must be an input place p of t that cannot be marked in $(\mathcal{N}, \emptyset)$. Consequently, $p \in U$, $t \in U^\bullet$, and ${}^\bullet U \subseteq U^\bullet$ hold such that U turns out to be a co-trap in contrast to the assumption. So, U must be empty. Then, all places are markable sufficiently high such that all transitions are firable arbitrarily often, i.e. $(\mathcal{N}, \emptyset)$ is live.

Corollary 1. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net;*

$(\mathcal{N}, \emptyset)$ is live and backward live iff $\mathcal{N}$ contains neither a co-trap nor a trap.

(The "backward liveness" of $(\mathcal{N}, M)$ is the liveness of $(\mathcal{N}', M)$ where $\mathcal{N}'$ arises form $\mathcal{N}$ by reversing the direction of all arcs.)

Proof. That is an immediate consequence of the fact that traps (co-traps) are transformed into co-traps (traps) if the direction of all arcs is changed.

This corollary guarantees for example that the reproduction condition (a t-invariant $r \geqq \emptyset$ exists whose net representation $\mathcal{N}_r$ neither contains a trap nor a co-trap) is necessary and sufficient for the liveness of $(\mathcal{N}_r, \emptyset)$ in both directions.

Corollary 2. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net;*

there exists a dead transition in $(\mathcal{N}, \emptyset)$ iff $\mathcal{N}$ contains a co-trap.

Proof. If $(\mathcal{N}, \emptyset)$ contains a dead transition, $(\mathcal{N}, \emptyset)$ is not live and thus contains a co-trap according to theorem 2.

If $\mathcal{N}$ contains a co-trap D, D is empty under $\emptyset$ such that all transitions $t \in D^\bullet$ are dead in $(\mathcal{N}, \emptyset)$.

Corollary 3. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net and $r \geqq \mathbf{0}$ a t-invariant of $\mathcal{N}$ such that the empty marking $\emptyset$ can be reproduced by realization of $k \cdot r$ for some $k \in \mathbb{N} \backslash \{0\}$;*

then the net representation $\mathcal{N}_r$ of r neither contains a trap nor a co-trap.

Proof. If $\emptyset$ can be reproduced by realization of $k \cdot r$ for $k \in \mathbb{N} \backslash \{0\}$, $(\mathcal{N}_r, \emptyset)$ is live in both directions. Then $\mathcal{N}_r$ contains no trap and no co-trap according to corollary 1.

This completes the necessity part of the proof of theorem 1.

It is more intricate to show that the reproduction condition of theorem 1 is also sufficient for the reproducibility of $\emptyset$. The basic idea for that part of the proof is an induction on the number of shared places. We therefore need some details about p/t-nets without shared places.

Lemma 5. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net without shared places;*

then $\mathcal{N}$ contains neither a trap nor a co-trap iff $\mathcal{N}$ is a synchronization graph without cycles.

Proof. Since $\mathcal{N}$ does not contain shared places $|{}^\bullet p| \leqq 1$ and $|p^\bullet| \leqq 1$ holds for all $p \in S$. If $\mathcal{N}$ does not contain traps and co-traps there exist no boundary places because if p is a boundary place $\{p\}$ is either a co-trap or a trap. So, $|{}^\bullet p| = |p^\bullet| = 1$ holds for all $p \in S$, i.e. $\mathcal{N}$ is a synchronization graph. According to lemma 3 $\mathcal{N}$ does not contain cycles.

If $\mathcal{N}$ is a synchronization graph without cycles $\mathcal{N}$ consists of chains beginning and ending with boundary transitions. Assume there is a co-trap D in $\mathcal{N}$ and $p \in D$. Since p is no input boundary place there is a chain $K = \{k_0, k_1, k_2, \ldots, k_{2l}, p\}$ such that $k_1, k_3, \ldots, k_{2l-1}, p$ are places of D and k_0 is an input boundary transition. But then $k_0 \in {}^\bullet D$ and $k_0 \notin D^\bullet$ in contrast to the assumption that D is a co-trap.

The next theorem and its corollary confirm that for p/t-nets without shared places the reproduction condition of theorem 1 is sufficient for reproducing the empty marking $\emptyset$.

Theorem 3. *Let $G = (S, T, F)$ be a synchronization graph without cycles and M a marking of G;*

then every integer $|T|$-vector $v \geqq \mathbf{0}$ is realizable in (G, M) iff

$$M \; + \; [G] \cdot v \geqq \mathbf{0}.$$

Proof. Let $v \geqq \mathbf{0}$ be realizable in (G, M), i.e. there exists a firing sequence σ with $M[\sigma\rangle (M + [G] \cdot v)$ s.t. all transitions t occur in σ as often as $v(t)$ indicates. As a marking $M + [G] \cdot v$ is non-negative.

Let now be $M + [G] \cdot v \geqq \mathbf{0}$. $v = \mathbf{0}$ is trivially realizable by firing no transition at all.

In case $v \geqq \mathbf{0}$ a transition $z_0 \in T$ exists with $v(z_0) > 0$. If z_0 is not enabled by M and thus not suited to be the first transition of a realization of v there exists a place $p_0 \in {}^\bullet z_0$ with $M(p_0) = 0$. In $M + [G] \cdot v \geqq \mathbf{0}$ one inequality belongs to p_0, namely $M(p_0) - v(z_0) + v(z_1) \geqq 0$ with $\{z_1\} = {}^\bullet p_0$. This implies $v(z_1) \geqq v(z_0) > 0$. If z_1 is not enabled for M, too, there is a transition z_2 with $v(z_2) > 0$ etc. By continuing thus one gets a backwards directed chain of transitions $z_0, \ldots, z_n$, where $v(z_i) > 0, 0 \leqq i \leqq n$. The chain is finite because G is finite and cycle-free. The fact that z_n is enabled for M is ensured because eventually the chain ends in an input boundary transition that is enabled trivially.

Let $y_1 \in T$ be a transition with $v(y_1) > 0$ that is enabled for M. Let furthermore $\chi(y_1)$ be the characteristic $|T|$-vector of y_1. Then

$$\begin{aligned} M + [G] \cdot v &= M + [G] \cdot \chi(y_1) + [G] \cdot (v - \chi(y_1)) \\ &= M + G(y_1) + [G] \cdot (v - \chi(y_1)) \geqq \mathbf{0} \end{aligned}$$

where $G(y_1)$ is the column of G belonging to y_1. $M_1 := M + G(y_1)$ is the follower marking of M after one firing of y_1. Then $v_1 := v - \chi(y_1)$ is that part of v that still must be realized. For $k := \sum_{t \in T} v(t)$ the following holds

$$\sum_{t \in T} v_1(t) \; = \; k - 1.$$

In case $k - 1 > 0$, the procedure has to be repeated for M_1 and $v_1 \ldots$. After k steps the algorithm ends with $v_k = 0$.

The sequence $\sigma = y_1 \ldots y_k$ that transforms M into M_k is a realization of v. Every transition y occurs in σ $v(y)$ times.

Corollary 4. *Let $G = (S, T, F)$ be a synchronization graph without cycles and let $r \geqq \mathbf{0}$ be a $|T|$-vector;*

then in G the empty marking $\emptyset$ can be reproduced by realizing r iff r is a t-invariant of G.

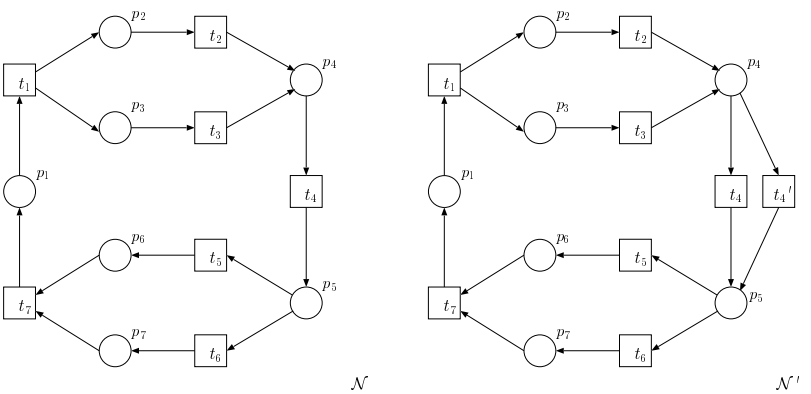
Proof. If $\emptyset$ can be reproduced by realizing r, r is a t-invariant.

If $r \gneqq \mathbf{0}$ is a t-invariant $\mathbf{0} + [G] \cdot r = \mathbf{0}$ holds. Due to theorem 3 r is realizable in $(G, \emptyset)$; it is a reproduction of $\emptyset$ in G.

A well known result ([Murata89], theorem 16) says that in an acyclic Petri net $\mathcal{N}$ B is reachable from A iff there exists a non-negativ integer solution x satisfying

$$B = A + [\mathcal{N}] \cdot x$$

This means that for synchronization graphs without cycles the existence of a t-invariant $r \gneqq \mathbf{0}$ is necessary and sufficient for the reproducibility of the empty marking. So far we get a proof of theorem 1 for p/t-nets without shared places. To start off handling p/t-nets with shared places we will concentrate on t-invariants because those parts of the nets which are not belonging to a t-invariant cannot contribute to a reproduction of markings. So, we will assume for some time that the nets are covered by a t-invariant $r > \mathbf{0}$. Moreover, we assume $r = \mathbf{1}$. This is no further restriction because in cases $r(t) = k_t \geqq 2$ $k_t - 1$ copies of t can be added to the original net thus getting a new net with equal behaviour in which $\mathbf{1}$ is a t-invariant. For example, in the following net $\mathcal{N}$ $r = (1, 1, 1, 2, 1, 1, 1)^t$ is a t-invariant. Adding one copy of transition t_4 results in the net $\mathcal{N}'$ where $\mathbf{1}$ is the t-invariant corresponding to r.



In nets with arc weights equal to 1, a further profitable consequence of this assumption is that $|{}^\bullet p| = |p^\bullet|$ holds for all places $p \in S$, that is every places has as many ingoing as outgoing arcs.

The next theorem shows that the reproduction condition of theorem 1 is sufficient if $\mathbf{1}$ is a t-invariant.

Theorem 4. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net that contains neither traps nor co-traps and let $\mathbf{1}$ be a t-invariant of N;*

then the empty marking is reproducible in $\mathcal{N}$ by realization of $k \cdot \mathbf{1}$ for some $k \in \mathbb{N} \setminus \{0\}$.

Proof. The proof is based on an induction on the number m of shared places. For $m = 0$ the statement of the theorem follows from corollary 4.

Supposing the statement is true for p/t-nets with $m-1$ shared places. Let p be a shared place of $\mathcal{N}$ and let $\mathcal{N}_{(-p)} = (S_{(-p)}, T, F_{(-p)})$ be the net that results from $\mathcal{N}$ by omitting p. Thus $S_{(-p)} := S\backslash\{p\}, F_{(-p)} := F\backslash((\{p\} \times T) \cup (T \times \{p\}))$. Then $\mathcal{N}_{(-p)}$ is a p/t-net for which the following holds:

(a) $\mathcal{N}_{(-p)}$ contains no trap and no co-trap since every trap (co-trap) of $\mathcal{N}_{(-p)}$ is a trap (co-trap) of $\mathcal{N}$.
(b) The $|T|$-vector $\mathbf{1}$ is also a t-invariant of $\mathcal{N}_{(-p)}$ because $[\mathcal{N}_{(-p)}]$ arises from $[\mathcal{N}]$ by omitting the row belonging to p.
(c) In $\mathcal{N}_{(-p)}$ the empty marking is reproducible by realizing $k_{(-p)} \cdot \mathbf{1}$ for some $k_{(-p)} \in \mathbb{N}\backslash\{0\}$ since $\mathcal{N}_{(-p)}$ has $m - 1$ shared places.

In a realization of $k_{(-p)} \cdot \mathbf{1}$ in $\mathcal{N}_{(-p)}$ every transition fires $k_{(-p)}$ times. Adding in $\mathcal{N}$ and $\mathcal{N}_{(-p)}$ to every transition $(k_{(-p)} - 1)$ copies of itself one obtains the nets $\mathcal{N}^* = (S, T^*, F^*)$ and $\mathcal{N}^*_{(-p)} = (S_{(-p)}, T^*, F^*_{(-p)})$, respectively.

In $\mathcal{N}^*$ the $|T^*|$-vector $\mathbf{1}$ is a t-invariant because $[\mathcal{N}^*] = [\underbrace{[\mathcal{N}] \ldots [\mathcal{N}]}_{k_{(-p)} \text{ times}}]$. Furthermore, $\mathcal{N}^*$ does not contain traps and co-traps as $\mathcal{N}$ does not; $(\mathcal{N}^*, \emptyset)$ is therefore live in both directions (corollary 1).

In $\mathcal{N}^*_{(-p)}$ the empty marking is reproducible by realizing the $|T^*|$-vector $\mathbf{1}$ (because of (c)), i.e. every $t \in T^*$ fires exactly once.

Let σ^* be a firing sequence in $\mathcal{N}^*_{(-p)}$ that realizes the $|T^*|$-vector $\mathbf{1}$, thus reproducing the empty marking. Of course, σ^* is not necessarily a firing sequence in $(\mathcal{N}^*, \emptyset)$ because there might be markings in which p is not sufficiently markable to enable transitions of $p^\bullet$. So p might be kind of a bottleneck in $(\mathcal{N}^*, \emptyset)$. Nevertheless, we will exploit σ^* to some extent for $(\mathcal{N}^*, \emptyset)$. That is, we will transform σ^* into another firing sequence σ' by reversing the order of concurrent subsequences such that σ' is also a realization of $\mathbf{1}$ in $\mathcal{N}^*_{(-p)}$ that reproduces the empty marking.

Whether σ^* is a firing sequence in $(\mathcal{N}^*, \emptyset)$ or not depends on the existence of cycles through p in $\mathcal{N}^*$. Clearly, if there don't exist cycles through p in $\mathcal{N}^*$ σ^* is a firing sequence that reproduces the empty marking in $\mathcal{N}^*$.

Let

$$T^\bullet(p) := \{t \in T^* | (\exists n \geqq 0 : (t,p) \in F^{2n+1}) \wedge (\exists p' \in t^\bullet : (\forall t' \in (p')^\bullet : t' \in T^\bullet(p)) \vee p' = p)\}$$

be the set of transitions whose firing causes necessarily a flow of tokens through p. Let

$${}^\bullet T(p) := \{t \in T^* | (\exists n \geqq 0 : (p,t) \in F^{2n+1}) \wedge (\exists p' \in {}^\bullet t : (\forall t' \in {}^\bullet(p') : t' \in {}^\bullet T(p)) \vee p' = p)\}$$

be the set of transitions which cannot be activated without tokens flowing through p.

Then

$$C(p) := T^\bullet(p) \cap {}^\bullet T(p)$$

is the set of transitions which necessarily transport tokens on cycles through p.

Next, we partition the transitions of ${}^\bullet p \cup p^\bullet$ w.r.t. their membership of $C(p)$:

$$D(p) := {}^\bullet p \cap C(p)\,,\; D'(p) := p^\bullet \cap C(p)$$
$$I(p) := {}^\bullet p \setminus C(p)\,,\; I'(p) := p^\bullet \setminus C(p)$$

Since $(\mathcal{N}^*, \emptyset)$ is live it is possible to lead up at least one token to p without firing a transition of $C(p)$. Consequently $I(p) \neq \emptyset$. Let σ_1 be a firing sequence in N^* with $\emptyset[\sigma_1\rangle$ where as many as possible transitions fire exactly once without any firing of a transition of $C(p)$. σ_1 which is also a firing sequence of $(\mathcal{N}^*_{(-p)}, \emptyset)$ is the first part of σ'. In other words, σ_1 arises form a subsequence of σ^* by rearranging concurrently enabled transitions.

Similarly, since $(\mathcal{N}^*, \emptyset)$ is also backwards live it is possible to lead up at least one token to p in backward direction without firing a transition of $C(p)$. So $I'(p) \neq \emptyset$. Let ϱ be the backward firing sequence in $(\mathcal{N}^*, \emptyset)$ starting at $\emptyset$ where as many transitions as possible fire exactly once without any firing of a transition of $C(p)$. Then ϱ in reverse direction less all transitions already occurring in σ_1 is the third (and last) part σ_3 of σ'. The middle part σ_2 of σ' is σ^* less all transitions already occurring in σ_1 and σ_3. $\sigma' = \sigma_1\sigma_2\sigma_3$ is a firing sequence of $(\mathcal{N}^*_{(-p)}, \emptyset)$ but not necessarily of $(\mathcal{N}^*, \emptyset)$ because σ_1 might not pump enough tokens into p to make $\sigma_2\sigma_3$ possible.

We now aim at finding a $k' \in I\!N \setminus \{0\}$ such that

$$k'\sigma' := (k'\sigma_1)(k'\sigma_2)(k'\sigma_3) := \underbrace{\sigma_1 \ldots \sigma_1}_{k'}\underbrace{\sigma_2 \ldots \sigma_2}_{k'}\underbrace{\sigma_3 \ldots \sigma_3}_{k'}$$

is a realization of $k' \cdot \mathbf{1}$ in $(\mathcal{N}^*, \emptyset)$ thus reproducing the empty marking.

Since $\mathbf{1}$ is a t-invariant of $\mathcal{N}^*$ $|{}^\bullet p| = |p^\bullet|$ holds. Moreover, because of $I(p) \cup D(p) = {}^\bullet p$ and $I'(p) \cup D'(p) = p^\bullet$ the following holds

$$k' \cdot |{}^\bullet p| = k'(|I(p)| + |D(p)|) = k'(|I'(p)| + |D'(p)|) = k'|p^\bullet|$$

for all $k' \in I\!N$ and

$$-|D'(p)| + |D(p)| = |I'(p)| - |I(p)|.$$

We distinguish two cases:

First case: $-|D'(p)| + |D(p)| \geqq 0$,

that is the token count on p caused by $(k'\sigma_1)$ is not decreased by $(k'\sigma_2)$. In order to get sufficiently many tokens on p by $(k'\sigma_1)$ to make $(k'\sigma_2)$ possible the restriction on k' is $k'|I(p)| \geqq |D'(p)|$ and

$$k' \geqq \tfrac{|D'(p)|}{|I(p)|}.$$

Such a k' exists because of $I(p) \neq \emptyset$.

Second case: $-|D'(p)| + |D(p)| < 0$,
i.e. the token count on p is decreased during $(k'\sigma_2)$. Then, after $(k'\sigma_1)((k'-1)\sigma_2)$ sufficiently many tokens have to be on p to make σ_2 once more possible. That means for the token count on p

$$\begin{aligned} |D'(p)| &\leqq k'|I(p)| \ + \ (k'-1)(-|D'(p)| \ + \ |D(p)|) \\ &= k'|I(p)| \ + \ (k'-1)(|I'(p)| \ - \ |I(p)|) \\ &= |I(p)| \ + \ (k'-1)|I'(p)| \\ k' &\geqq \frac{|D'(p)| \ - \ |I(p)|}{|I'(p)|} \ + 1 \end{aligned}$$

Such a k' exists because of $I'(p) \neq \emptyset$.

Altogether,

$$k' \geqq \max \left\{1, \frac{|D'(p)|}{|I(p)|}\right\} \text{ in first case}$$

$$k' \geqq \max \left\{1, \frac{D'(p)| \ - \ |I(p)|}{|I'(p)|} \ + 1\right\} \text{in second case}$$

For the original p/t-net $\mathcal{N}$ the empty marking is reproducible by realization $k_{(-p)} \cdot k' \cdot \mathbf{1}$.

Example 1. In the net of figure 1 the following holds:

$$\begin{aligned} T^\bullet(p_1) &= T \setminus \{t_B\} \\ {}^\bullet T(p_1) &= T \setminus \{t_A\} \\ C(p_1) &= T^\bullet(p_1) \cap {}^\bullet T(p_1) \ = \ T \setminus \{t_A, t_B\} \\ D(p_1) &= {}^\bullet p_1 \cap C(p_1) \ = \ \{t_1, t_2\} \\ D'(p_1) &= p_1^\bullet \cap C(p_1) \ = \ \{t_4, t_5\} \\ I(p_1) &= {}^\bullet p_1 \setminus C(p_1) \ = \ \{t_A\} \\ I'(p_1) &= p_1^\bullet \setminus C(p_1) \ = \ \{t_B\} \end{aligned}$$

So we find

$$\begin{aligned} |I(p_1)| = |I'(p_1)| \ &= \ 1 \\ |D(p_1)| = |D'(p_1)| \ &= \ 2 \\ -|D'(p_1)| + |D(p_1)| = 0 \quad &\text{(first case)} \\ k = \max \left\{1, \ \frac{2}{1}\right\} \ &= \ 2 \end{aligned}$$

is the minimal possible multiple.

$$\begin{aligned} (2\sigma) &= (2\sigma_1)(2\sigma_2)(2\sigma_3) \\ &= (t_A t_A)(t_5 t_4 t_3 t_2 t_1 \ t_5 t_4 t_3 t_2 t_1) \ (t_B t_B) \end{aligned}$$

reproduces the empty marking.

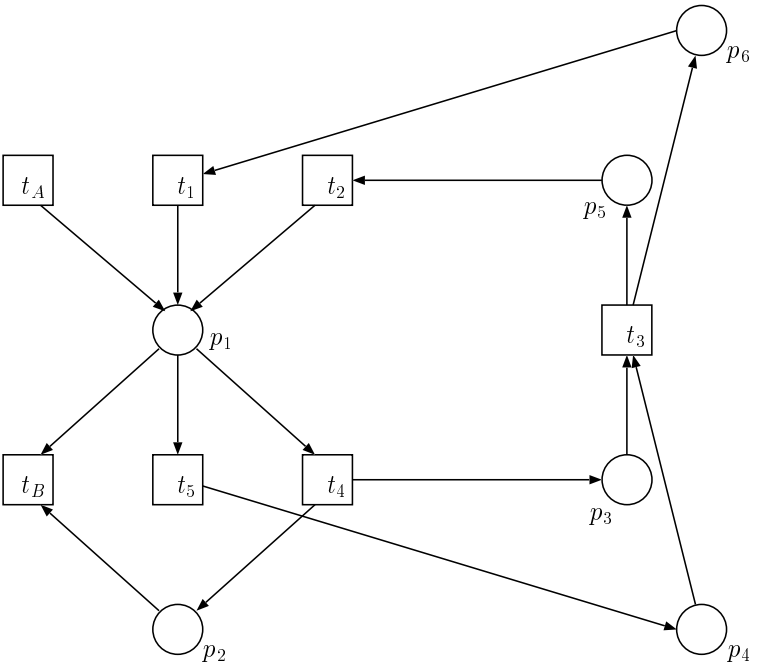


Fig. 1.

Example 2. In the net of figure 2 the following holds

$$
\begin{aligned}
T^\bullet(p_1) &= T\backslash\{t_B\} \\
{}^\bullet T(p_1) &= T\backslash\{t_A, t_1\} \\
C(p_1) &= T^\bullet(p_1) \cap {}^\bullet T(p_1) = T\backslash\{t_A, t_B, t_1\} \\
D(p_1) &= {}^\bullet p_1 \cap C(p_1) = \{t_2\} \\
D'(p_1) &= p_1^\bullet \cap C(p_1) = \{t_4, t_5\} \\
I(p_1) &= {}^\bullet p_1 \setminus C(p_1) = \{t_A, t_1\} \\
I'(p_1) &= p_1^\bullet \setminus C(p_1) = \{t_B\}
\end{aligned}
$$

So we find

$$
\begin{aligned}
|I(p_1)| = |D'(p_1)| &= 2 \\
|I'(p_1)| = |D(p_1)| &= 1 \\
-|D'(p_1)| + |D(p_1)| &= -1 \quad \text{(second case)} \\
k = \max\left\{1, \frac{2-2}{1} + 1\right\} &= 1
\end{aligned}
$$

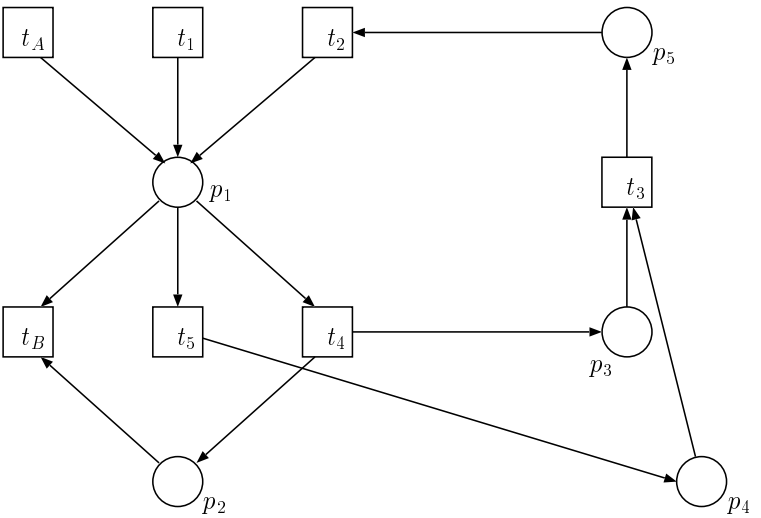


Fig. 2.

is the minimal possible multiple.

$$\begin{aligned}\sigma &= (\sigma_1)(\sigma_2)(\sigma_3)\\ &= (t_A t_1)(t_5 t_4 t_3 t_2)\ (t_B)\end{aligned}$$

reproduces the empty marking.

Proof (of Theorem 1). Let $r \gneqq \emptyset$ be a t-invariant and let $\mathcal{N}_r = (S_r, T_r, F_r)$ be the corresponding net representation that does not contain a trap or a co-trap. Then the net $\mathcal{N}^* = (S_r, T_r^*, F_r^*)$, that arises from $\mathcal{N}_r$ by adding $(r(t) - 1)$ copies of every transition $t \in T_r$, does also not contain a co-trap or a trap. Moreover, the $|T^*|$-vector $\mathbf{1}$ is a t-invariant of $\mathcal{N}_r^*$.

Due to theorem 4 in $\mathcal{N}_r^*$ the empty marking is reproducible by realization of $k \cdot \mathbf{1}$ for some $k \in \mathbb{N} \setminus \{0\}$. Then in $\mathcal{N}_r$ the empty marking is reproducible by a corresponding realization of $k \cdot r$, i.e. whenever in $\mathcal{N}_r^*$ a copy of $t \in T_r$ fires in $\mathcal{N}_r$ t fires itself. So, the reproduction condition is sufficient. Its necessity is an immediate consequence of corollary 3.

The section will be concluded by two corollaries.

Corollary 5. *Let $\mathcal{N} = (S, T, F)$ be a p/t-net, $r \gneqq \emptyset$ a t-invariant, and $\mathcal{N}_r$ its net representation;*

then the empty marking is reproducible by realization of $k \cdot r$ for some $k \in \mathbb{N} \setminus \{0\}$ iff $\mathcal{N}_r$ is live in both directions.

Proof. Immediate consequence of theorem 4 and corollary 1.

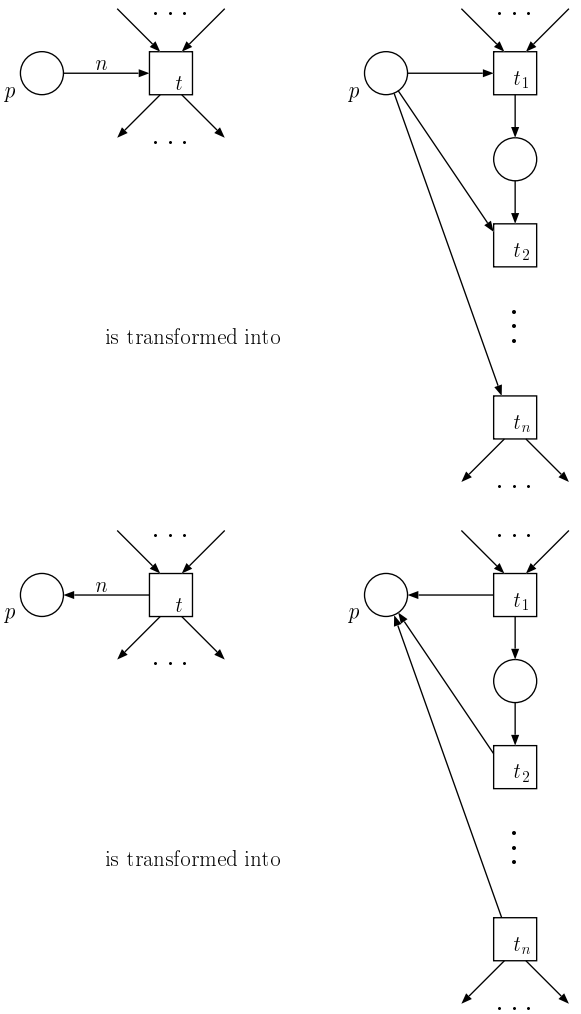
The next corollary extends the result of this section to p/t-nets of the form $\mathcal{N} = (S, T, F, W)$.

Corollary 6. *Let $\mathcal{N} = (S, T, F, W)$ be a p/t-net and $r \geqq \emptyset$ a t-invariant of $\mathcal{N}$;*

then in $\mathcal{N}$ the empty marking is reproducible by realizing $k \cdot r$ for some $k \in \mathbb{N} \setminus \{0\}$ iff the net representation $\mathcal{N}_r$ of r neither contains a trap nor a co-trap.

Proof. If in $\mathcal{N}$ the empty marking is reproducible by realization of $k \cdot r$ $\mathcal{N}_r$ neither contains a trap nor a co-trap.

If $\mathcal{N}_r$ neither contains a trap nor a co-trap the following transformation of $\mathcal{N}_r$ into a p/t-net $\mathcal{N}'_r$ whose arc labels all are equal to 1 does not alter that property.



In $\mathcal{N}'_r$ a t-invariant r' exists as a counterpart to r. Since in $\mathcal{N}'_r$ the empty marking can be reproduced by realizing $k \cdot r'$ this is possible in $\mathcal{N}_r$ by realizing $k \cdot r$.

4 Conclusion

In this paper a necessary and sufficient condition for the reproducibility of the empty marking was introduced. In essence, the condition states that the net representations of the relevant T-invariants should have neither traps nor co-traps.

References

1. J. M. Couvreur, Ezpeleta and M. Silva. *A new technique for finding a generating family of siphons, traps and st-components. Application to colored Petri Nets.* Advances in Petri Nets, 1993.
2. F. Commoner. *Deadlocks in Petri nets.* Report #CA-7206-2311, Applied Data Research, Inc., Wakefield, 1972.
3. T. Murata. *Petri Nets: Properties, Analysis and Applications.* Proceedings 77(4), IEEE, 1989.
4. J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
5. W. Reisig. *Petri Nets.* EATCS Monographs on Theoretical Computer Science vol. 4. Springer Verlag, New York, 1985.

Modeling and Analysis of Multi-class Threshold-Based Queues with Hysteresis Using Stochastic Petri Nets

Louis-Marie Le Ny[1] and Bruno Tuffin[2]

[1] IRISA-Université de Rennes 1, Campus universitaire de Beaulieu
35042 Rennes cedex, France
leny@irisa.fr
[2] IRISA-INRIA, Campus universitaire de Beaulieu
35042 Rennes cedex, France
btuffin@irisa.fr

Abstract. This paper deals with multi-class queueing systems where thresholds are included in order to smooth the variations of throughput and delay by modifying the queue behaviour. Hysteresis is also inserted, so that the control mechanism will not switch too much. One motivation for using multiple classes of customers is its capability to model heterogeneous traffics like data, voice and video. Moreover, threshold queues have many applications in the transport protocols of communication networks. The analysis is done using Stochastic Petri Nets and Fluid Stochastic Petri Nets. This powerful paradigm helps to obtain a very simple representation of the systems and the analysis is transparent using an available Petri net package. Numerous numerical illustrations are given in order to validate the use of threshold queues with hysteresis as well as their representation by SPNs and FSPNs and performances of various scheduling schemes are compared in order to minimize a cost function.

Keywords: Hysteresis, Threshold queues, Performance analysis, Stochastic Petri Nets, Fluid Stochastic Petri Nets.

1 Introduction

We consider in this paper threshold-based queues with hysteresis and mutiple classes of customers. This kind of queueing system has many applications in the dynamic control of computer systems and telecommunication networks. For instance, congestion control and traffic shaping is a big challenge in DiffServ, the service differentiation which will drive the future Internet. Indeed, different quality of services (in terms of throughput, delay or jitter for instance) are required for different applications such as video, telephony, email, ftp... Our modelling will help to approach these characteristics. Of course, other applications of multi-class threshold queues with the same control principles can be found, like in [2] where it is applied to the signaling system no. 7.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 254–272, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Basically, the queues are controlled by a sequence of forward thresholds and a sequence of backward thresholds such that when the content of the queue exceeds a forward threshold, the services of some classes are speeded up. In the same way, when a backward threshold is reached, the services are slowed down. Due to the existence of different types of traffic, each queue is split into several independent queues, one for each type, and the server is allocated to one of these queues according to a given policy (some policies will be described later). Some delay may happen, for example when some management is necessary before modifying the parameter of the queue services.

Mono-class threshold-based queueing systems have been extensively studied in the literature, without hysteresis first in [17,21,23] and with hysteresis next [16,18,20,22]. In the most general setting, the queue is Markovian and a closed-form solution for the steady-state probabilities of a heterogeneous multi-server threshold queue with hysteresis is obtained. The works we are going to describe now are devoted to multi-class queues. In [19], different scheduling policies at an ATM switch are studied; packet loss probabilities and mean waiting times are compared in terms of the traffic load. In [1], threshold policies for a single-server queueing network with two job classes are used in order to devise and implement different policies. Numerical results, obtained with the power series algorithm, show that the model satisfies the requirements of higher-priority traffic while offering acceptable performance for the lower priority one. In [2], Choi *et al.* analyse a multi-class priority queueing system with congestion based on thresholds. Each class is controlled by a triple of thresholds composed of an abatement threshold, an onset threshold and a discard threshold. By then, a priority policy is included in the queue in order to provide a better quality of service to some classes. An algorithm to compute performance measures is given if the packet arrival process is a queue-length dependent Markovian arrival process. In [25], a quantitative study of differentiated services for the Internet (DiffServ) is given; the authors compare the loss and delay behaviors related to two router mechanisms called threshold dropping and priority scheduling. Using an analytical method, they show that a significant improvement in throughput can be achieved, especially when using priority scheduling. Finally, we quote a recent work on multi-class threshold-based queuing system with hysteresis: in [11], the number of servers is controlled by the threshold sequences and a fast and accurate iterative solution for evaluating the system performance under varying thresholds and parameters is given.

Since the above literature shows that an analytic analysis becomes quickly tricky, an alternative way, using Stochastic Petri Nets (SPNs), is attempted in this paper. Indeed, SPNs are proven to be a powerful representation of systems with synchronisation or concurrency and we show that the representation of threshold-based queues is very simple and convenient. When the SPN is Markovian, software packages can provide automated generation and solution of continuous-time Markov chains, otherwise it can be simulated. The general class of SPN we will use is actually Stochastic Reward Nets (SRNs) [4,6] where each state can be associated to a reward rate and allow some functionalities such as

guard functions (to enable or not events), state-dependence of firing times and events and priorities. We will also use a generalization of SRNs, Fluid Stochastic Petri Nets (FSPNs) [7,15,27] which is a hybrid extension of SPNs where fluid can model continuous components or approximate discrete states to prevent the explosion of the state-space.

We are going to model our systems by SPNs and their extensions like we have done in the mono-class case in [28]. This work can be related to [13], where a somehow similar multi-class analysis using SPNs has been done but targeted to polling systems.

To analyse our SPN and FSPN models, we are going to use one of the available software packages, SPNP (for *Stochastic Petri Net Package*) [5,14], a software developed at Duke University. This tool uses analytic-numeric methods to solve Markovian nets and simulate FSPNs and non-Markovian SPNs.

The organization of this paper is the following. Section 2 presents the systems we are going to model and analyse. Section 3 is devoted to a description of SPNs (in a very general setting) and FSPNs and Section 4 presents the description of multi-class threshold queues by means of SPNs and FSPNs. Numerical analysis is provided in Section 5. Finally, we give some conclusions and directions for future work in Section 6.

2 Model Definition

We consider here multi-class systems, wich are required when dealing with different QoS levels. We consider a set of L different classes arriving to a one-server queue. The queue capacity is assumed to be finite, with capacity C_l for class-l (but a common capacity can be devised as well). A state of the system is given by the number n_l $(1 \leq l \leq L)$ of class-l customers in the system and the class number of the customer in service. In this model, each class is assumed to have a separate buffer. Nevertheless, the queue is not separate in several independent queues (even if it can be) because the service policy depends on the global state of the queue. We can affect for instance a strict priority at each class, but many other policies can be devised. We follow here the general policy of [19] where two-class queues are analysed and where a class-l customer will be served with probability $f_l(i,j)$ when there are i class-1 customers and j class-2 customers. The policies analyzed in [19] are

1. Head of line priority (HOL): $f_1(i,j) = 1$ if $i > 0$.
2. Bernoulli scheme (BS): $f_1(i,j) = p$ and $f_2(i,j) = 1 - p$ (if $i, j > 0$).
3. Queue Length Threshold scheduling (QLT): class-1 customers have strict priority ($f_1(i,j) = 1$) if $j < T$, i.e., if the number of class-2 customers is under a threshold. Above this level, class-2 has strict priority.
4. Queue length threshold on class-2 buffer with Bernoulli scheduling (QB2): A threshold value T is used for the number of queued class-2 customers. We use $f_1(i,j) = 1$ if $i > 0$ and $j < T$ and $f_1(i,j) = p$ if $i > 0$ and $j \geq T$.

5. Queue length threshold on class-1 buffer with Bernoulli scheduling (QB1): the threshold value is now used for class-1 queue. We use $f_1(i,j) = p$ if $0 < i < T$ and $j > 0$ and $f_1(i,j) = 1$ if $i \geq T$ and $j > 0$.

An exact analysis is done in [19] (with two classes) when the model is Markovian. This is very general because policies like head of line priority, Bernoulli scheduling scheme or queue length threshold can be represented using functions f_l as we will see below. This model can easily be generalized to more than two classes.

Indeed, we can define $K-1$ forward and backward multi-dimensional thresholds, defined over

$$\{(n_1, \cdots, n_L) \ : 0 \leq n_l \leq C_l\},$$

having a partial order (even if in the subsequent application the order is actually total) and such that when a threshold is reached, the probability functions $f_l(n_1, \cdots, n_L)$ are modified. We have K probability functions corresponding to the $K-1$ forward thresholds: when the k^{th} forward threshold is reached, probability functions are switched to $f_l^{(k+1)}(\cdot)$ and when the k^{th} backward threshold is reached, probability functions are switched back to $f_l^{(k)}(\cdot)$. It is then a generalization of the work in [19]. Hysteresis comes from the fact that forward and backward thresholds do not have the same values. Then, the number of costful switches (in management operations) is expected to decrease.

The way we introduce hysteresis in the policies of [19] is the following (with $K = 2$):

1. HOL: incorporating hysteresis to HOL scheduling would lead to behave like when incorporating hysteresis to QB1 or QB2, so we do not consider it here.
2. BS: the two thresholds s_1 (backward) and S_1 (forward) depend on the number of class-1 customers in the queue. We take $f_1^{(1)}(i,j) = p_1$ and $f_1^{(2)}(i,j) = p_2$ with $p_2 > p_1$ so that if class-1 queue has a lot of customers, we increase the probability that this class is served.
3. QLT: thresholds depend here only on class-2 buffer content. Two thresholds s_2 (backward) and S_2 (forward) are introduced such that $f_1^{(1)}(i,j) = 1$ and $f_1^{(2)}(i,j) = 0$. It means that until S_2 is reached class-1 has strict priority. Then strict class-2 priority is used until s_2 is hit.
4. QB2: thresholds depend here only on class-2 buffer content. We use $f_1^{(1)}(i,j) = 1$ (if $i > 0$) and $f_1^{(2)}(i,j) = p$ (if $i > 0$). We then have two thresholds instead of one in [19] and the probability function used (when between those thresholds) depends on the last threshold hit.
5. QB1: thresholds depend here only on class-1 buffer content. We use $f_1^{(1)}(i,j) = p$ if $i,j > 0$ and $f_1^{(2)}(i,j) = 1$ if $j > 0$ and the scheme works like QB2.

Inter-arrivals and services will first be assumed to be exponential. Even if this exponential assumption is valid for some applications such as telephone networks, many application areas need more general distribution assumptions. In the ATM and Internet networks for instance, services should be assumed to be

determistic; moreover, arrivals often present self-similar or heavy-tailed behavior (see for instance [8]). We will study these non-Markovian threshold-queues in the following. As another generalization, there can be some delay between the time that the threshold is reached and the time that the parameters are changed. This can happen for instance when the queue monitors its content at fixed or random intervals of time. These cases will also be investigated numerically.

3 Stochastic Petri Nets, Stochastic Reward Nets and Fluid Stochastic Petri Nets

Petri nets is a formal graph particularly well suited for representing the flow of information and control in systems with concurrency and synchronization characteristics. We describe here an extension of SPNs and SRNs which we will abusively call this model SPN. We then define FSPNs.

The class of stochastic Petri Nets we are considering is given by a tuple

$$(P, T, D, D^0, g, m^0, >, d, \omega, r, F)$$

where

- P is the set of places (represented by circles). Each place may contain *tokens*. The *marking* of the SPN is then defined by the number of tokens in each place. Let $\mathcal{S}$ be the set of markings
- T is the set of transitions (represented by bars).
- $D : (((P \times T) \cup (T \times P)) \times \mathcal{S}) \to \mathbb{N}$ is the set of input arcs (from a place to a transition) and output arcs (from a transition to a place). Each arc is given by its (marking-dependent) multiplicity.
- $D^0 : (P \times T \times \mathcal{S}) \to \mathbb{N}$ is the set of inhibitor arcs, from a place to a transition (represented by an arc terminated by a small circle), with its associate multiplicity.
- $g : (T \times \mathcal{S}) \to \{0, 1\}$ is the (marking-dependent) guard function for each transition. It is a generalization of inhibitor arcs, nevertheless we still also consider inhibitor arcs for their graphical usefulness and for sake of generality.
- $m^0 \in \mathbb{N}^{|P|}$ is the initial marking.
- $>: T \to \mathbb{N}$ is the explicit priority to transitions.
- $d : T \to \{\text{distributions}\}$ defines the firing time distribution of each transition (note that it includes immediate transitions).
- $\omega : (T \times \mathcal{S}) \to \mathbb{R}^+$ is a weight function used to choose the one which will fire if several have the same firing time.
- $r : T \to \{\text{policies}\}$ is the static resampling policy for each transition when it becomes enabled again after being disabled by the firing of a concurrent transition. Three policies are possible: PRI (preemptive repeat identical), PRD (preemptive repeat different) and PRS (preemptive resume). The default value is PRD

– $F : (T \times T) \rightarrow \{\text{policies}\}$ defines the affecting resampling policy for each transition when another transition fires but the considered transition remains enabled. PRI, PRS and PRD are also possible and the default value is PRS.

The dynamics of SPNs is the following. A transition is said to be *enabled* if each of its input places contains as many tokens as the multiplicity of the corresponding arc and if the next two properties are verified. First the *inhibitor arcs* (if any), must verify that each inhibitor place must contain a number of tokens less than the multiplicity of the corresponding arc. The second required property is that the value of the guard function of the transition (if any) is 1. Among all enabled transitions, the one with the smallest firing time is said to *fire*. The firing times of transitions follow random distributions which can be general (i.e., not only exponential). If several transitions have the same firing time, the one which will actually fire is sampled with a probability equal to its individual weight divided by the sum of weights of all enabled transitions. When it fires, the transition removes a number of tokens from each of its input places equal to the multiplicity of the corresponding arc and it deposits in each of its output place a number of tokens equal to the multiplicity of the corresponding arc, leading then to a new marking.

An FSPN [7,15,27] is an extension of an SPN where some places can be fluid and where some fluid can flow through transitions, following given rules. Fluid places are represented by two concentric circles and fluid arcs by directed thick arcs. The dynamics of an FSPN is then governed by discrete events (firing of transitions modifying the discrete and fluid marking) and continuous flows in fluid places between these firings as described in the following. When the transitions related to fluid arcs (input or output) are enabled, fluid flows along the arc with the constraint that the fluid level remains between 0 and a prespecified bound. For computational simplicity, the fluid flow rate is at most linear: for each fluid place p, the fluid level $x_p(t)$ changes according to the equation [7]

$$\frac{dx_p(t)}{dt} = a(m)x_p(t) + b(m)$$

where m is the current discrete marking and a and b are some functions of the discrete marking. Moreover, when a transition fires, some fluid can be immediately put in or removed from fluid places. The arcs related to these marking-dependent fluid impulses of the input and output connecting transitions and fluid places are represented by thin directed arcs like the arcs connecting transitions and discrete places. Of course, the initial marking x^0 in fluid places, needs also to be specified.

4 Representation by Means of SPNs and FSPNs

4.1 Markovian Queue

Consider a two-class system, where, for instance, class-1 is for demanding service (such as video or real-time traffic) and class-2 for less stringent service

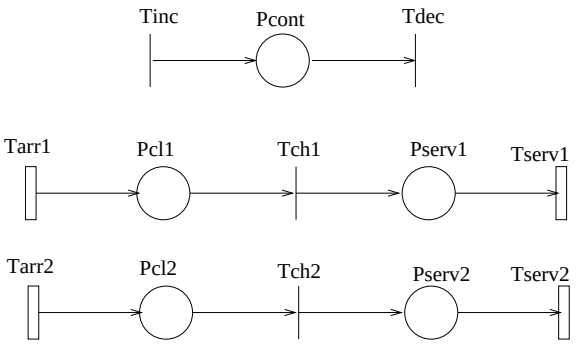


Fig. 1. Representation of a two-class threshold-queue with hysteresis by an SPN

(like e-mail,...). It seems better to give a higher service probability (or a strict priority) to the class-1 customers, especially when there are many class-1 customers waiting in the buffer. Figure 1 represents such a system by a Markovian SPN with rate functions and guard functions described in Tables 1 and 2. A generalization to more classes is straightforward. The number of tokens place $Pcont$ gives the control integer k such that probability functions $f_l^{(k)}(\cdot)$ are used (which, in Petri Net formalism, are actually the weight functions). Of course, $f_1^{(k)}(n_1, n_2) + f_2^{(k)}(n_1, n_2) = 1$ $(k = 1, 2)$ $\forall n_1, n_2$. Transitions $Tinc$ and $Tdec$ fire only when forward and backward thresholds are hit. In this example, thresholds depend only on class 1 queue but it can easily be made more general (by modifying the guard functions of transitions $Tinc$ and $Tdec$ in Table 2).

Transitions $Tarr1$ and $Tarr2$ are for arrivals of class-1 and class-2 customers respectively (of course their rates can be made dependent of an external stochastic process like On/Off sources). The guards on these transitions are such that there is a separate buffer for each class (of respective capacity $C1$ and $C2$). This can be easily changed to a shared buffer by setting the guard function of $Tarr1$ and $Tarr2$ to $\#Pcl1 + \#Pcl2 < C$. Places $Pcl1$ and $Pcl2$ are for class-1 and class-2 customers waiting to be served. Places $Pserv1$ and $Pserv2$ give the class of the customer in service. Only one token can be in $Pserv1$ and $Pserv2$ (exclusively), this being controlled by using the guard on transitions $Tch1$ and

Table 1. rate functions of the SPN of Figure 1

Transition	Rate
$Tarr1$	λ_1
$Tarr2$	λ_2
$Tserv1$	μ
$Tserv2$	μ

Table 2. Guard functions of the SPN of Figure 1

Transition	guard
$Tinc$	$\#Pcl1 + \#Pserv1 > S_{\#Pcont}$ && $\#Pcont < K$
$Tdec$	$\#Pcl1 + \#Pserv1 \leq s_{\#Pcont-1}$ && $\#Pcont > 0$
$Tarr1$	$\#Pcl1 < C1$
$Tarr2$	$\#Pcl2 < C2$
$Tch1$	$\#Pserv1 == 0$ && $\#Pserv2 == 0$
$Tch2$	$\#Pserv1 == 0$ && $\#Pserv2 == 0$

Table 3. Weight functions of the SPN of Figure 1

Transition	probability (or weight)
$Tch1$	$f_1^{(\#Pcont)}(\#Pcl1 + \#Pserv1, \#Pcl2 + \#Pserv2)$
$Tch2$	$f_2^{(\#Pcont)}(\#Pcl1 + \#Pserv1, \#Pcl2 + \#Pserv2)$

$Tch2$ giving the class to be served. When the customer is served (transitions $Tserv1$ and $Tserv2$), the new customer class to be served is chosen using the probabilities/weights $f_l^{(k)}(n_1, n_2)$ as defined in Table 3.

4.2 General Queue

In the non-Markovian case, marking-dependent exponential distributions of transitions $Tarr1$, $Tarr2$, $Tserv1$ and $Tserv2$ are replaced by marking-dependent general distributions. We introduce two places $Pentrance1$ and $Pentrance2$ in order to model arrivals (admitted or not) and to be able to compute loss probabilities (because PASTA property does not apply anymore). The model is described in Figure 2. With respect to the model of the previous subsection, rates ar replaced by general distributions, and we introduce guard functions for arrivals at the queue in Table 4, meaning that the new token/customer/packet is admitted if the queue is not full. These guard functions can be modified in order to apply admission control.

Table 4. Supplementary Guard functions of the SPN of Figure 2

Transition	guard
$Tloss1$	$\#Pcl1 == C$
$Tok1$	$\#Pcl1 < C$
$Tloss2$	$\#Pcl2 == C$
$Tok2$	$\#Pcl2 < C$

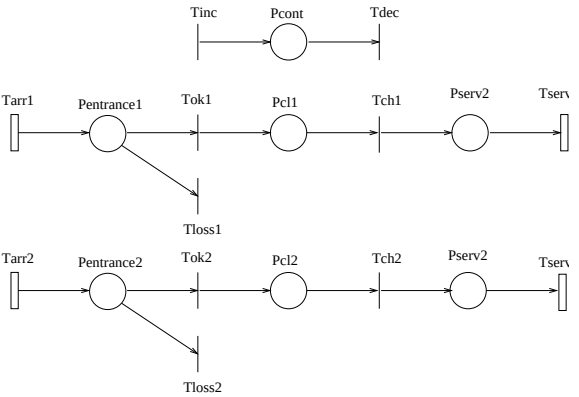


Fig. 2. Queue with general distributions

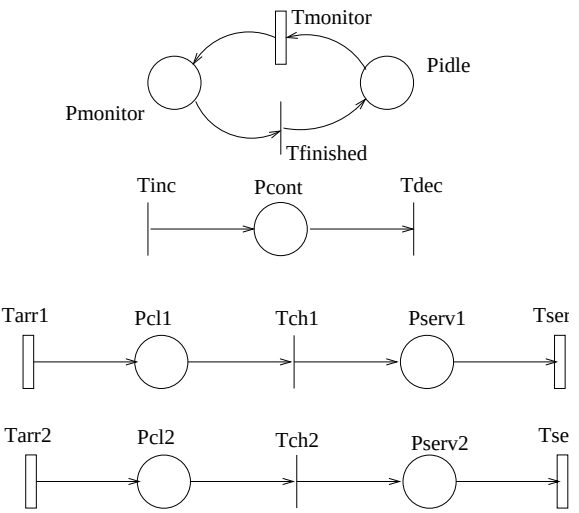


Fig. 3. A queue with delay

4.3 Queue with Management Delay

Delay can be introduced when the system monitors the state of the queue only at fixed or random instants of time. Thus, the policy is modified only when the monitoring is done. Figure 3 represents such a queue. The system behaves like in the non-delayed case but supplementary conditions are introduced on the guard functions of transitions $Tinc$ and $Tdec$ which represent the switches in the service policy. These two transitions are enabled only if the monitoring has just

Table 5. Guard functions of the SPN of Figure 3

Transition	guard
$Tinc$	$\#Pcl1 + \#Pserv1 > S_{\#Pcont}$ && $\#Pcont < K$ && #Pmonitor==1
$Tdec$	$\#Pcl1 + \#Pserv1 \leq s_{\#Pcont-1}$ && $\#Pcont > 0$ && $\#Pmonitor == 1$
$Tarr1$	$\#Pcl1 < C1$
$Tarr2$	$\#Pcl2 < C2$
$Tch1$	$\#Pserv1 == 0$ && $\#Pserv2 == 0$
$Tch2$	$\#Pserv1 == 0$ && $\#Pserv2 == 0$

been done, i.e., if there is a token in place $Pmonitor$. Then, immediate transitions $Tinc$ and $Tdec$ can fire (if the other conditions on the guard functions are fulfilled). Next, transition $Tfinished$ fires so that the system waits for another period of time to monitor the queue. Among possible concurrent transitions, a higher priority is given to transitions $Tinc$ and $Tdec$ with respect to $Tfinished$ to ensure that the service modification is applied before place $Pmonitor$ becomes idle again. The new guard functions are given in Table 5. Other parameters are defined like in the Markovian non-delayed case, but we can also generalize the case where inter-arrivals and/or services do not follow an exponential distribution.

4.4 Fluid Queue

We consider here a fluid queue driven by *On-Off* sources, but other driving processes can be used as well. Discrete arrivals and services are aggregated in a continuous flow. This is a good approximation of ATM traffic for instance. The FSPN modelling this system is described in Figure 4 with guard functions and fluid rate functions described respectively in Tables 6 and 7. The arrival fluid rates are made dependent of the number of *On* sources for each class. If the buffer is full (i.e., $\#Psyst1 + \#Psyst2 = C$), these rates are also bounded by the current service rate (i.e., the remaining flow is lost). The service fluid rates are such that the proportion p and $1-p$ of served class-1 and class-2 fluid depend on the hysteretic control variable like in the discrete case. Here, if a place is empty, the fluid arrival rate is bounded by the service fluid rate. Also, fluid rates are optimized in such a way that if the buffer is empty for a given class, the other class is served instead.

Table 6. Guard functions of the SPN of Figure 4

Transition	guard
$Tinc$	$\#Psyst1 > S_{\#Pcont}$ && $\#Pcont < K$ && $r(Tarr1) > r(Tdep1)$
$Tdec$	$\#Psyst1 \leq s_{\#Pcont-1}$ && $\#Pcont > 0$ && $r(Tarr1) < r(Tdep1)$

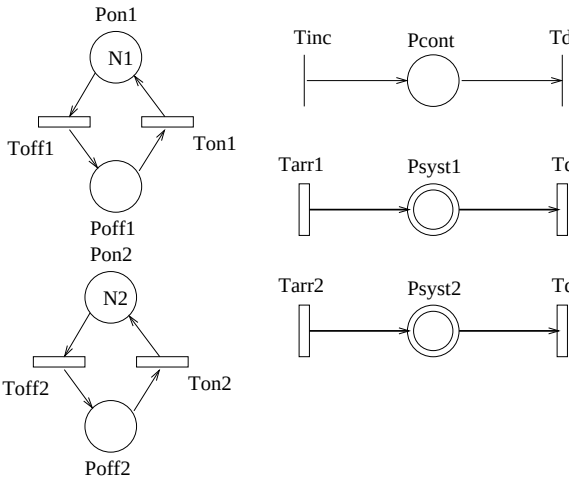


Fig. 4. Two-class fluid queue

We have made the thresholds dependent of the class-1 fluid level, but we can make it dependent of the class-2 fluid level or of a linear combination of both as well.

5 Analysis

5.1 Solution Methods

The solution methods we are going to use are those implemented in SPNP [5,14], a package developed at Duke University, and are the following.

Markovian SPNs of the previous sections are solved analytically-numerically. The reachability graph is first generated from the SPN and then converted to a Markov chain (or more exaclty to a Markov reward model) which is solved numerically. For steady-state evaluation, we will choose Steady-State SOR (Successive Overrelaxation [26]), the fastest method, but Steady-State Gauss-Seidel [26], and Steady-State Power method [30] are also avaible in SPNP and are used when SSOR does not converge. For transient-state solution of the CTMC, standard uniformization or uniformization using the Fox and Glynn method for computing the Poisson probabilities can be used.

On the other hand, non-Markovian SPNs and FSPNs are solved by simulation (because it is currently the only method implemented in SPNP). Nevertheless, there exists some numerical analysis algorithms solving FSPNs when there are one (see for instance [15]) or at most two fluid places in the system [31]. Steady-state simulation will be performed using simulation with batches or regenerative

Table 7. Fluid rate functions of the SPN of Figure 4

Transi.	Fluid rate r(Transition)
$Tarr1$	$\lambda 1 \#Pon1\ 1_{[\#Psyst1+\#Psyst2<C]} +$ $\min(\lambda 1 \#Pon1, r(\text{Tdep1}))1_{[\#Psyst1+\#Psyst2=C]}$
$Tarr2$	$\lambda 2 \#Pon2\ 1_{[\#Psyst1+\#Psyst2<C]} +$ $\min(\lambda 2 \#Pon2, r(\text{Tdep1}))1_{[\#Psyst1+\#Psyst2=C]}$
$Tdep1$	$\mu p(\#Pcont)$ if $(\#Psyst1 > 0$ and $\#Psyst2 > 0)$ $\min(\lambda_1 \#Pon1, \mu p(\#Pcont))$ if $(\#Psyst1 = 0$ and $\#Psyst2 > 0)$ $\mu - \min(\lambda_2 \#Pon2, \mu(1 - p(\#Pcont)))$ if $(\#Psyst1 > 0$ and $\#Psyst2 = 0)$ $\min(\lambda_1 \#Pon1, \mu - \min(\lambda_2 \#Pon2, \mu(1 - p(\#Pcont))))$ if $(\#Psyst1 = 0$ and $\#Psyst2 = 0)$
$Tdep2$	$\mu(1 - p(\#Pcont))$ if $(\#Psyst2, \#Psyst1 > 0)$ $\min(\lambda_2 \#Pon2, \mu(1 - p(\#Pcont)))$ if $(\#Psyst2 = 0,\ \#Psyst1 > 0)$ $\mu - \min(\lambda_1 \#Pon1, \mu p(\#Pcont))$ if $(\#Psyst2 > 0,\ \#Psyst1 = 0)$ $\min(\lambda_2 \#Pon2, \mu - \min(\lambda_1 \#Pon1, \mu p(\#Pcont)))$ if $(\#Psyst2 = 0$ and $\#Psyst1 = 0)$

simulation [9] (in this last case, our SPN needs to be in the class of MRSPNs [3]). Transient simulation can be performed by crude independent replications. However, there are other simulation methods available in SPNP: importance splitting techniques (Restart and splitting) [29], importance sampling [9,10,24], regenerative simulation with importance sampling [12,24], all suitable to estimate rare events.

5.2 Numerical Results

We consider a two-class queue like in Figure 1. We assume that class-1 traffic is for real-time applications and is more important than class-2 traffic. The queue capacities are $C_1 = C_2 = 64$; the thresholds are $s_1 = 24$ and $S_1 = 48$ in the hysteresis case and $S = (S_1 = s_1 =)36$ if the policy does not require hysteresis. The cost function we use here is divided in two parts, the network cost and the user cost. The user cost function, representing what is felt by the user application, is

$$C_{user} = c_1 P(\text{Class-1 loss}) + c_1' P(\text{Class-2 loss}) + c_2 P(\text{queue 1 empty}) + c_3 g_1(m_1) + c_3' g_2(m_2)$$

where function $g_1(m_1)$ for class-1 depending on the number m_1 of class-1 customers in the queue is hyperbolic:

$$g_1(m) = \begin{cases} \frac{64}{3m+16} & \text{if } m \leq 16, \\ \frac{64}{3(64-m)+16} & \text{if } m \geq 48, \\ 1 & elsewhere. \end{cases}$$

By then, the queue is penalized if it is close to be full or close to starvation. Function $g_2(m_2)$ for class-2 has an hyperbolic arc only when the class-2 buffer

is close to be full (if it is empty, it does not matter). The network cost function, representing the additional costs in order to improve the QoS without any perception by the user, is

$$C_{net} = c_4 \mu P(\text{server used}) + c_5 \alpha_u + c_6 \alpha_d$$

where α_u (resp. α_d) is the mean number of forward switches (resp. backward switches) per unit time when changing of policy (i.e., of values f_k). Note that $\alpha_u = \alpha_d$ in steady state. For our examples, we arbitrarily choose the weights to be $c_1 = 40$, $c_1' = 4$, $c_2 = 15$, $c_3 = 0.8$, $c_4 = 0.05$, $c_5 = 0.4$, $c_6 = 0.4$ and $c_3' = 0.2$. Moreover, the service rate is $\mu = 10.0$ and the total arrival rate is λ so that $\lambda_1 = \lambda r$ and $\lambda_2 = \lambda(1 - r)$ where r is the fraction of traffic allocated to class-1. As we are interested in steady-state analysis, the initial marking specification is not necessary.

Consider first a Markovian model. The (exact) results are displayed by means of 3D-surfaces; in this way, one can easily see the lowest cost and hence choose the best policy with respect to the parameters p and r. Only the schemes showing the best performance are displayed in order to keep the surfaces clear. Recall that p denotes the probability for serving a class-1 customer first and r is the proportion of class-1 arriving customers. For comparison sake, in the case of hysteresis, we take $p_1 = p$ and $p_2 = p + 0.2$; therefore class-1 is more likely to be served if its buffer has passed the forward threshold. We are going to observe the behavior of the schemes when the total traffic increases, i.e., for different values of λ.

In Figure 5 the Bernoulli scheduling with and without hysteresis are investigated under light traffic ($\rho = 0.3$) for which they are giving the best results. The costs are computed for $p = 0.3$, $p = 0.5$ and $p = 0.7$ whereas the value of r ranges from 0.1 to 0.9 with a step of 0.1. The surfaces are obtained using the cubic spline method available in the MATLAB package. Figure 5 shows that the best policy is BS (Bernoulli scheduling without hysteresis) under some conditions on the values of r and p:

- the ratio r of class-1 customers is high (beyond 0.8).
- r is between 0.4 and 0.8, and p is close to 0.5.
- $r < 0.4$ and $p > 0.2$.

In the other cases, the policy BShy (Bernoulli scheduling with hysteresis) performs better. If one wants to pull out a simple and approximative rule, we can suggest to use BS if $p < r$ and BShy otherwise. Note that higher costs for small values of r (and λ) are due to queue starvation which happens often, due to the light traffic.

Taking $\rho = 0.5$ (see Figure 6) the same two policies give the best results. The rule (using BS if $p < r$ and BShy otherwise) is still valid since the frontier is clearer. In heavier traffic (Figure 7), for $\rho = 0.8$, the best policies are Bernoulli Scheduling with hysteresis (BShy) or QB1 with hysteresis (QB1hy) depending on the values of p and r. If p is low (less than 0.3), for any value of r, the less expensive policy is QB1hy, whereas if p is higher the cost values are almost the

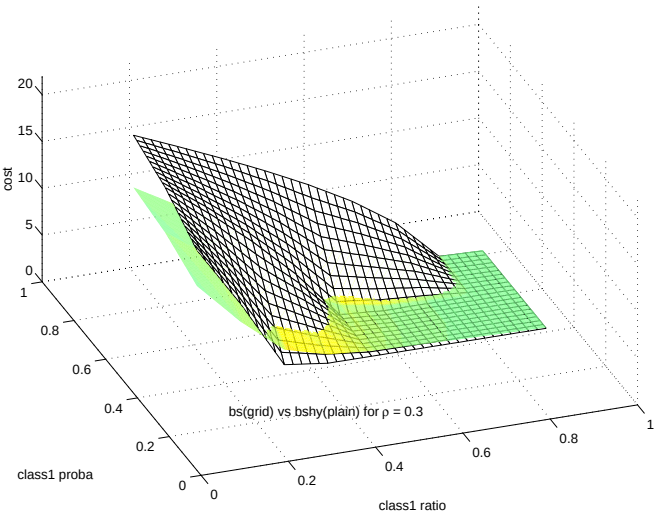


Fig. 5. Cost comparisons between policies Bernoulli Scheduling (BS) and BS with Hysteresis (BShy) when $\rho = 0.3$

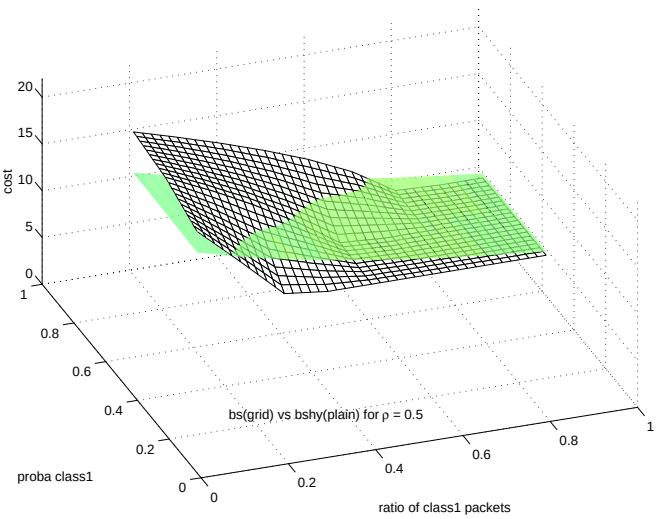


Fig. 6. Cost comparisons between policies Bernoulli Scheduling (BS) and BS with Hysteresis (BShy) for $\rho = 0.5$

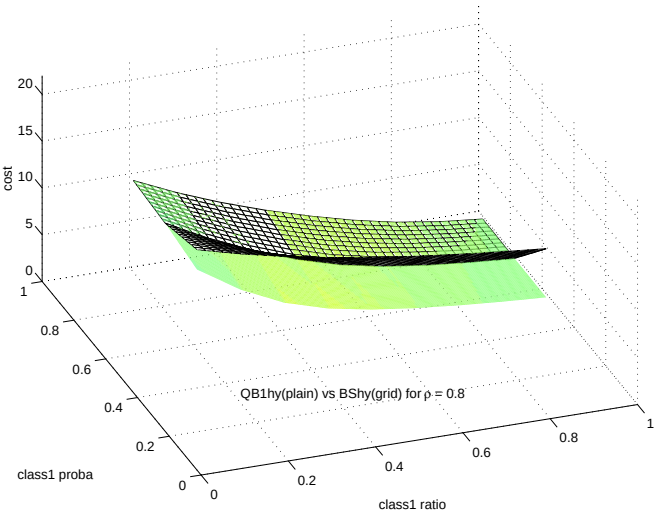


Fig. 7. Cost comparisons between policies Bernoulli Scheduling with Hysteresis (BShy) and QB1 with Hysteresis (QB1hy) when $\rho = 0.8$

Table 8. Optimal policies with respect to r, p, and ρ for the Markovian queue

ρ	0.3	0.5	0.8	1.1
the best policy	BS $(p < r)$ BShy $(p > r)$	BS $(p < r)$ BShy $(p > r)$	QB1hy	QB1hy $(p > r)$ BShy $(p < r)$

same and it is thus possible to keep the QB1hy policy. Finally, in Figure 8, we consider a congestion case, i.e., $\rho = 1.1$. Again, we almost have that, if $p < r$, the best policy is BS with hysteresis whereas if $p > r$, QB1 with hysteresis performs better. All the above results are summarized in Table 8. Of course, this rule is valid only for our cost function and numerical values, but it shows how a rule can be displayed using our modeling in order to choose the best service policy with respect to traffic parameters.

We now study non-Markovian queues. The parameters used are deterministic services with value 5.0, inter-arrivals following Pareto distributions (with shape parameter 0.05/2.1 and smallest value 0), so that the total traffic intensity is $\rho = 0.5$. We give in Table 9 numerical results showing that the previous Markovian rule does not apply here as QB1hy performs the best. It means that another rule has to be devised by running the model when the parameters varies.

Table 10 shows, using QB1hy scheduling of the previous non-Markovian queue that delay can slightly modify the performance of the system (compared with the same policy in Table 9). We are using here an inter-monitoring time interval following an exponential distribution with rate 1.0, the other parameters being exactly the same than in the above general queue.

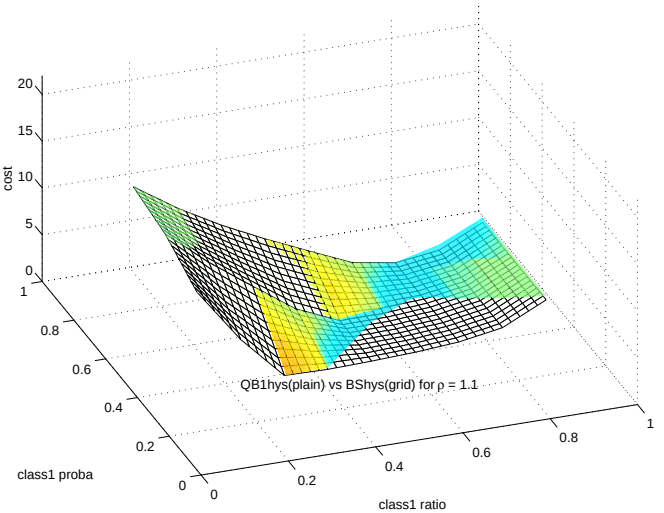


Fig. 8. Cost comparisons between policies Bernoulli Scheduling with Hysteresis (BShys) and QB1 with Hysteresis (QB1hys) when $\rho = 1.1$

Table 9. Results obtained for the general queue when $\rho = 0.5$ and $r = 0.6$

Scheme	user cost	network cost	total cost
HOL	[40.670,42.477]	[0.19742,0.20194]	[40.870,42.677]
BS, $p = 0.3$	[54.210,56.897]	[0.18022,0.18436]	[54.392,57.080]
BS, $p = 0.7$	[47.816,50.008]	[0.18356,0.18763]	[48.001,50.194]
QLT	[67.670,70.987]	[0.19487,0.19939]	[67.867,71.185]
QB1, $p = 0.7$	[41.514,43.351]	[0.21275,0.21791]	[41.729,43.567]
QB2, $p = 0.7$	[54.197,56.562]	[0.22117,0.22590]	[54.420,56.786]
BShy, $p = 0.3$	[56.479,59.065]	[0.20213,0.20617]	[56.683,59.270]
BShy, $p = 0.7$	[48.879,50.961]	[0.21360,0.21758]	[49.094,51.177]
QLThy	[64.239,67.430]	[0.19020,0.19461]	[64.431,67.623]
QB1hy, $p = 0.7$	[38.449,40.093]	[0.19392,0.19824]	[38.645,40.289]
QB2hy, $p = 0.7$	[44.439,46.513]	[0.18991,0.19436]	[44.631,46.706]

Consider now the system as a fluid queue. Table 11 displays the results obtained for QB1hy scheduling when $\rho = 0.5$ and $p = 0.5$. The parameters are here $C_1 = C_2 = 2.0$, $p = 0.5$, the service fluid rate is $\mu = 4.4$; there are 10 On-Off sources for each class, the fluid arrival rate being $\lambda = 0.4$ per source for both classes. The thresholds are $S_1 = 1.5$, $s_1 = 0.5$. Note that the expression of costs have to be modified in order to fit the fluid modelling. The loss probability is computed as the ratio of the mean feeding fluid rate $\lambda E(\#Pon)$ minus the mean of $r(Tarr)$ on $\lambda E(\#Pon)$ (and the empty probability is computed using the same principle). The hyperbolic cost is the one which has to undergo numer-

Table 10. Results obtained for the general queue with delay when $\rho = 0.5$ and $r = 0.6$ using QB1hy scheduling

Scheme	user cost	network cost	total cost
QB1hy $p = 0.7$	[40.904,42.733]	[0.18983,0.19422]	[41.096,42.926]

Table 11. Results obtained for the fluid queue using QB1hy scheduling

Scheme	user cost	network cost	total cost
QB1hy $p = 0.5$	[5.2649,5.6235]	[5.5849e-01,5.9072e-01]	[5.8353,6.2023]

ous adaptations. We precomputed the different possibilities of integrals of $g_1(\cdot)$ and $g_2(\cdot)$ between two instants of time and added them to the cost at each step of the discrete event simulation. The hyperbolic curves are on intervals $[0, 0.4]$ and $[1.6, 2]$.

6 Conclusion

Modeling and analysis of threshold-based queues using SPNs and FSPNs is very convenient and SPNP tool (for instance) allows to do it quickly. This can aid in the study and configuration of the queue to conform to design specifications. Multi-class queues are important due to their applications in the next Internet generation which must include service differentiation. We have modeled those queues in several situations (Markovian, non-Markovian, with delay, fluid) and for several service policies between classes. By varying parameters, the model can specify the best schedulling policy. Moreover, if the network knows the traffic intensity at different times of day, it can apply different policies in order to improve the performance.

As directions for future work, we can enhance:

- Test and devise protocols where the nodes of the network notify to the sources when and how to speed-up or slow down their transmission rates.
- Using Colored SPNs and devise Colored Fluid Stochastic Petri Nets in order to separate the flows of different sources and to analyse the behaviour of each traffic stream. Moreover, color will be a more natural way to model multi-class systems.

Acknowledgement

The authors would like to thank the anonymous reviewers for their suggestions and comments.

References

1. K. D. Ansell, P. S. Glazebrook and I. Mitrani. Threshold policies for a single-server queuing network. *Probability in the Engineering and Informational Sciences*, 15:15–33, 2001. 255
2. B. D. Choi, S. H. Choi, B. Kim, and D. K. Sung. Analysis of priority queueing system based on thresholds and its application to signaling system no. 7 with congestion control. *Computer Networks*, 32:149–170, 2000. 254, 255
3. H. Choi, V. G. Kulkarni, and K. S. Trivedi. Markov Regenerative Stochastic Petri Nets. *Performance Evaluation*, 20(1-3):337–357, 1993. 265
4. G. Ciardo, A. Blakemore, P. F. Chimento, J. K. Muppala, and K. S. Trivedi. Automated Generation and Analysis of Markov Reward Models using Stochatic Reward Nets. In Carl Meyer and Robert Plemmons, editors, *Linear Algebra, Markov Chains and Queuing Models*, volume 48 of *IMA Volumes in Mathematics and its Applications*, pages 145–191. Springer-Verlag, Heidelberg, 1993. 255
5. G. Ciardo, J. K. Muppala, and K. S. Trivedi. SPNP: Stochastic Petri net Package. In *Proc. Third International Workshop on Petri Nets and Performance Models, PNPM'89*, pages 142–151, 1989. 256, 264
6. G. Ciardo, J. K. Muppala, and K. S. Trivedi. Analyzing Concurrent and Fault-tolerant Software Using Stochatic Reward Nets. *Journal of Parallel and Distributed Computing*, 15:255–269, 1992. 255
7. G. Ciardo, D. M. Nicol, and K. S. Trivedi. Discrete-Event Simulation of Fluid Stochastic Petri-Nets. *IEEE Transactions on Software Engineering*, 25(2):207–217, 1999. 256, 259
8. M. E. Crovella. Performance Characteristics of the World Wide Web. In G. Haring et al., editor, *Performance Evaluation*, volume 1769 of *Lecture Notes in Computer Science*, pages 219–232. Springer-Verlag, 2000. 258
9. G. S. Fishman. *Monte Carlo: Concepts, Algorithms and Applications*. Springer-Verlag, 1997. 265
10. P. W. Glynn and D. L. Iglehart. Importance Sampling for Stochastic Simulations. *Management Science*, 35(11):1367–1392, November 1989. 265
11. L. Golubchik and J. C. S. Lui. A fast and accurate iterative solution of a multi-class threshold-based queuing system with hysteresis. In *Proc. of the ACM SIGMETRICS Conference*, pages 196–206, Santa Clara, June 2000. 255
12. A. Goyal, P. Shahabuddin, P. Heidelberger, V. F. Nicola, and P. W. Glynn. A Unified Framework for Simulating Markovian Models of Highly Dependable Systems. *IEEE Transactions on Computers*, 41(1):36–51, January 1992. 265
13. B. R. Haverkort. *Performance of Computer Communication Systems*. John Wiley and Sons, 1998. 256
14. C. Hirel, B. Tuffin, and K. S. Trivedi. SPNP Version 6.0. In B. R. Haverkort, H. C. Bohnenkamp, and C. U. Smith, editors, *Computer performance evaluation: Modelling tools and techniques; 11th International Conference; TOOLS 2000, Schaumburg, Il., USA*, volume 1786 of *Lecture Notes in Computer Science*, pages 354–357. Springer Verlag, 2000. 256, 264
15. G. Horton, V. Kulkarni, D. Nicol, and K. S. Trivedi. Fluid Stochastic Petri nets: Theory, Application and Solution. *European Journal of Operational Research*, 105:184–201, 1998. 256, 259, 264
16. O. C. Ibe and J. Keilson. Multi-server threshold queues with hysteresis. *Performance Evaluation*, 21:185–213, 1995. 255

17. R. L. Larsen and A. K. Agrawala. Control of a Heterogeneous Two-Server Exponential Queueing System. *IEEE Transactions on Software Engineering*, SE-9(4):522–526, 1983. 255
18. L-M. Le Ny and B. Tuffin. A simple analysis of heterogeneous multi-server threshold queues with hysteresis. Technical Report 1333, IRISA, 2000. 255
19. J. Y. Lee and Y. H. Kim. Performance analysis of a hybrid priority control scheme for input and output queueing ATM switches. In *Proceedings of IEEE INFOCOM 98*, pages 1470–1477, March 1998. 255, 256, 257
20. S. Q. Li. Overload Control in a Finite Message Storage Buffer. *IEEE Transactions on Communications*, 37(12), December 1989. 255
21. W. Lin and P. R. Kumar. Optimal Control of a Queueing System with Two Heterogeneous Servers. *IEEE Transactions on Automatic Control*, AC-29(8):696–703, 1984. 255
22. J. C. S. Lui and L. Golubchik. Stochastic complement analysis of multi-server threshold queues with hysteresis. *Performance Evaluation*, 35:185–213, 1999. 255
23. J. A. Morrison. Two-server queue with one server idle below a threshold. *Queueing Systems: Theory and Applications*, 7:325–336, 1990. 255
24. V. F. Nicola, M. K. Nakayama, P. Heidelberger, and A. Goyal. Fast Simulation of Highly Dependable Systems with General Failure and Repair Processes. *IEEE Transactions on Computers*, 42(12):1440–1452, December 1993. 265
25. D. Sahu, S. Towsley and J. Kurose. A Quantitative Study of Differentiated Services for the Internet. *Journal of Communications and Networks*, 2:127–137, 2000. 255
26. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994. 264
27. K. S. Trivedi and V. G. Kulkarni. FSPNs: Fluid Stochastic Petri Nets. In *14th International Conference on Applications and Theory of Petri Nets*, pages 24–31, 1993. 256, 259
28. B. Tuffin and L-M. Le Ny. Modeling and analysis of threshold queues with hysteresis using stochastic Petri nets: the monoclass case. In *Proceedings of Petri Nets and Performance Models*, pages 175–184. IEEE CS Press, 2001. 256
29. B. Tuffin and K. S. Trivedi. Implementation of importance splitting techniques in stochastic Petri net package. In B. R. Haverkort, H. C. Bohnenkamp, and C. U. Smith, editors, *Computer performance evaluation: Modelling tools and techniques; 11th International Conference; TOOLS 2000, Schaumburg, Il., USA*, volume 1786 of *Lecture Notes in Computer Science*, pages 216–229. Springer Verlag, 2000. 265
30. W. B. van den Hout. *The Power-Series Algorithm: A Numerical Approach to Markov Processes*. Tilburg University, 1996. 264
31. K. Wolter. *Performance and Dependability Modelling with Second Order Fluid Stochastic Petri Nets*. PhD thesis, Technische Universitat Berlin, 1999. 264

Tackling the Infinite State Space of a Multimedia Control Protocol Service Specification

Lin Liu[1,2] and Jonathan Billington[1]

[1] Computer Systems Engineering Centre, University of South Australia
SCT Building, Mawson Lakes Campus, Mawson Lakes, SA 5095, Australia
[2] School of Computer and Information Science, University of South Australia
liuly002@students.unisa.edu.au
jonathan.billington@unisa.edu.au

Abstract. Coloured Petri Nets (CPNs) are used to model the service provided by an International Standard for the control of multimedia communications over telecommunication networks including the Internet, known as the Capability Exchange Signalling (CES) protocol. The state space of the CPN model includes all of the possible sequences of user observable events, known as the service language, which is a useful baseline against which the protocol can be verified. However, the CES service CPN possesses an infinite state space, due to unbounded communication channels. We parameterize the CPN with the channel capacity, propose and prove a recursive formula for its state space and provide an algorithm for its construction. The algorithm generates the state space for capacity l, from the state space for capacity $l-1$, providing incremental state space generation rather than generating a new state space for each value of l. The state space is linear in the size of the channel.

Keywords: Multimedia Networks and Protocols, Service Definitions, Parameterization, Coloured Petri Nets, Incremental State Space Algorithms.

1 Introduction

The Capability Exchange Signalling (CES) protocol is a sub-protocol of the H.245 standard [14]. H.245 is the "Control protocol for multimedia communication" developed by the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T). This recommendation provides the mechanism for multimedia communication parties to conduct negotiations on various issues.

Different parties involved in a multimedia call may have different transmit and/or receive capabilities. For example, one may be able to transmit and receive both audio and video streams, but others may only have the capability to receive and transmit audio information. So they need to make their capabilities known to each other. Then the multimedia streams sent by one party can be understood appropriately by its peer(s). The CES protocol is used by one multimedia party to inform a peer of its capabilities.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 273–293, 2002.
© Springer-Verlag Berlin Heidelberg 2002

From the point of view of protocol engineering, it is desirable that the properties of a protocol can be verified before implementation. Petri nets have proved to be a suitable formal method for this purpose [7,8,15,19]. A methodology for verifying protocols using Petri nets is presented in [6]. However, no work has been done on the verification of H.245 with formal methods.

The CES protocol is chosen as the first sub-protocol of H.245 to be investigated. Some results have been presented in [17,18]. While this previous work has focussed on analysing the operation of the CES protocol, this paper investigates its *service*. The service specification [12] of a protocol describes the capability provided by the protocol and its underlying medium (collectively known as the service provider) to its users. In contrast to a protocol definition which specifies how the protocol operates, a service specification is an abstraction of the behaviour of the service provider as viewed by its users. It does not consider the internal behaviour of the protocol. A service specification provides a baseline for protocol design. Once a protocol is defined, it should be verified against its service specification to make sure that it has met the user requirements encapsulated in the service.

We need to generate the *service language* [6] for verifying the protocol against its service. The service language is the set of all possible global sequences of *service primitives*, the abstract representations of the interactions between the service provider and the service user at their boundary. Our approach [6] to generating the CES service language is as follows. Firstly we model the CES service with CPNs, where each service primitive is modelled by one or more CPN transitions. Then we generate the Occurrence Graph (OG) of the CPN model. The OG is treated as a Finite State Automaton (FSA). This FSA can be minimised [4] to obtain a FSA which records all the possible sequences of service primitives, i.e. the service language.

Unfortunately the service of the CES protocol is not completely defined in H.245. According to the conventions for defining protocol services [12], the major deficiency of the CES service definition is that the end-to-end correlation of CES service primitives is not defined. We define this correlation based on the conventions and our understanding of the CES protocol. The CES service CPN model then is based on this completed service definition.

The CES protocol belongs to the class of transaction protocols. For many transaction protocols, each transaction can be considered as independent (e.g. [10]), therefore we only need to model the service of a single transaction. However, the transactions of the CES protocol are dependent. Therefore we have to consider multiple transactions in our CPN model so that all the possible sequences of primitives can be captured.

Our general CPN model of the CES service includes places which allow for unbounded data structures, modelling that the service provider includes a distributed network which will have arbitrary storage capacity for the data received from its users. This, unfortunately results in an infinite state space.

There is significant work on infinite state systems [1,2,3,5,15,16,20,21]. However, the emphasis of these studies is on verification methods for infinite systems.

The basic idea of these verification methods is firstly to model an infinite system in an abstract or symbolic way, with finite states, which hopefully preserves the properties of the original infinite system. Then the properties of the infinite systems can be verified by analysing the properties of the corresponding finite state models. However, the goal of this paper is *not* the verification of CES protocol properties. Instead, we are trying to obtain all the possible occurrence sequences of the CES service primitives (i.e. the service language) from the OG of the CES service CPN. This requires a representation of the infinite OG, so that the sequences can be obtained from it. As the next step, we will verify the CES protocol against its service based on the service language. Therefore, the results from previous studies are not applicable to the work presented in this paper.

In this paper we limit the maximum queue length representing the capacity of the communication channel to an arbitrary but finite value, l, and parameterize the CPN model with l. We then determine a recursive formula for the OG of the parameterized CPN. This result is significant because it makes it possible to obtain the CES service language for any finite capacity of the communication channel. We generate the OG when the channel capacity is one ($l = 1$), a trivial computation for CPN support tools such as Design/CPN [11], and then apply the recursive formula.

The paper is structured as follows. Section 2 describes the CES service, which is modelled with CPNs in Section 3. This model is validated in section 4. Section 5 develops the recursive formula, its proof, and an algorithm for constructing the OG for a given l. Finally, Section 6 summarises the results and points out future research directions.

2 The CES Service

In this section, we define the CES service based on the service definition of the CES protocol given in H.245. According to the conventions [12] and referring to an example service definition [13], our definition of the CES service comprises two parts:

- the definition of the CES service primitives, including their parameters; and
- the definition of the sequences of the CES service primitives which includes the sequences of primitives at one CES endpoint (i.e. local sequences) and the relationship between service primitives at two CES endpoints.

The definition of the CES service primitives and their local sequences are adopted from H.245, with modification where they are not consistent with the conventions. The end-to-end correlation of the CES service primitives is missing from H.245, and is defined in this section using Time Sequence Diagrams.

2.1 Overview of the CES Service

As we mentioned in Section 1, the CES protocol is used by one multimedia communication party to inform its peer of its capabilities. Figure 1 shows the

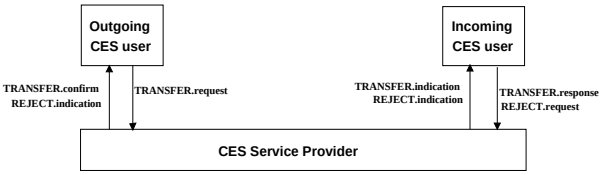


Fig. 1. The general model of the CES service

general model of the CES service, which comprises the CES users communicating over the CES service provider, by issuing service primitives. We only consider two peer users in the model, an outgoing CES user that initiates capability exchanges, and an incoming CES user that responds to the exchange requests.

The outgoing CES user initiates a capability exchange by issuing a TRANSFER.request primitive which includes its multimedia capabilities. At the incoming side, the service provider informs the user of the request with a TRANSFER.indication primitive. The incoming user indicates its acceptance of the request by issuing a TRANSFER.response primitive or its rejection with a REJECT.request primitive. In the first case, the outgoing user is notified by the service provider with a TRANSFER.confirm primitive, which concludes a successful capability exchange. In the second case, the outgoing user is informed with a REJECT.indication primitive. The service provider may reject a capability exchange by issuing a REJECT.indication primitive to the outgoing user and/or the incoming user.

2.2 The CES Service Primitives

Table 1, taken from H.245 [14], defines the CES service primitives and their parameters. The four TRANSFER primitives are used to transfer capabilities. The two REJECT primitives are used to reject a capability exchange, and to terminate a capability transfer. Primitives TRANSFER.request and TRANSFER.indication have the same parameters. PROTOID specifies the version of the recommendation in use. MUXCAP indicates the multiplexing capabilities of the outgoing end, and the multimedia receive and transmit capabilities are given by CAPTABLE and CAPDESCRIPTORS. The CAUSE parameter of a REJECT primitive indicates the reason for rejecting a CAPTABLE or CAPDESCRIPTORS parameter. The SOURCE parameter of the REJECT.indication indicates the source of the rejection, either USER or PROTOCOL.

2.3 Sequences of CES Service Primitives

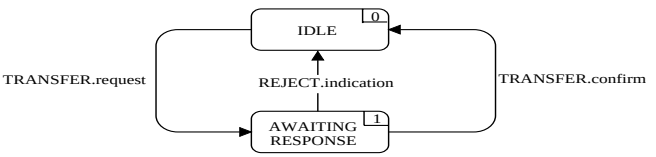
Local Sequences The allowed sequences of primitives at the outgoing and incoming ends are defined in H.245 as two separate state transition diagrams [14], which we include as Fig. 2. In H.245, the states of the CES protocol entities are used to define the local sequences, which is strictly not correct according to the conventions [12]. In our definition, instead, the states of the interface

Table 1. Primitives and parameters

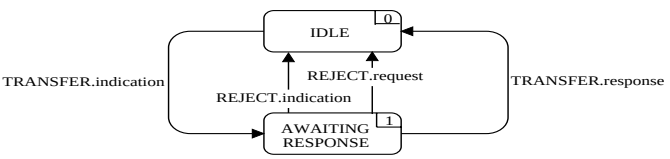
Generic name	Type			
	request	indication	response	confirm
TRANSFER	PROTOID MUXCAP CAPTABLE CAPDESCRIPTORS	PROTOID MUXCAP CAPTABLE CAPDESCRIPTORS	- (Note 1)	-
REJECT	CAUSE	SOURCE CAUSE	not defined (Note 2)	not defined
NOTE 1 - "-" means no parameters. NOTE 2 - "not defined" means that this primitive is not defined.				

between the CES service user and provider are used. There are two states defined for the outgoing interface: IDLE (state 0, indicating the interface is idle); and AWAITING RESPONSE (state 1, the interface is waiting for a response from its peer). The same state names are used for the incoming interface: IDLE (state 0, the interface is idle); and AWAITING RESPONSE (state 1, the interface is waiting for a response from the incoming user).

End-to-End Correlation of the CES Service Primitives As previously described, the outcome of a capability exchange request can be successful, or rejected by a CES user or the service provider. Figure 3(a) defines the sequence of primitives for a successful exchange. Figure 3(b) shows the sequence when the request of the outgoing user is rejected by the incoming user (REJECT.request). Figures 3(c) to 3(f) list four cases of provider initiated rejection. In Fig. 3(c), the incoming user has accepted the request by issuing a TRANSFER.response primitive, but the provider terminates the exchange before the response is passed to the outgoing user. Figure 3(d) is similar, only in this case the incoming user issues a rejection response. In Fig. 3(e) the incoming side does not receive any



(a) State transition diagram for sequences of primitives at outgoing end



(b) State transition diagram for sequences of primitives at incoming end

Fig. 2. Sequences of primitives at outgoing and incoming ends

information about the request. This may occur when the protocol message carrying the request is lost. Figure 3(f) specifies the situation when the service provider initiates rejection at both ends, resetting the system without waiting for the incoming user to generate its response. These two REJECT.indication primitives may or may not have a causal relationship.

It seems that Fig. 3 includes all the sequences of primitives for a capability exchange transaction. These sequences are consistent with the local sequences of the primitives in Fig. 2. However, from Fig. 2 we can see that a new TRANSFER.request can be issued before the previous one has been finished, if a REJECT.indication occurs at the outgoing end. It is possible for this new request to arrive at the incoming side when the incoming CES user is processing a previous request. The new request normally is an update of the old one, so the provider informs the incoming user to stop processing the previous request (REJECT.indication), and to take the new one (TRANSFER.indication). The REJECT.indication here is not considered as a provider initiated rejection, it is caused by the new request from the outgoing user, and the two exchange transactions are related.

Figures 4(a) to 4(e) include the sequences for this special user rejection. The difference between each figure is the way in which the second transaction finishes. It may be successful (Fig. 4(a)), rejected by the incoming user (Fig. 4(b)), or rejected by the provider (Figs. 4(c) and 4(e)), or by both (Fig. 4(d)). Based on Figs. 3 and 4, we are now able to generalise these sequences to any number of transactions, in accordance with the local sequences specified in Fig. 2.

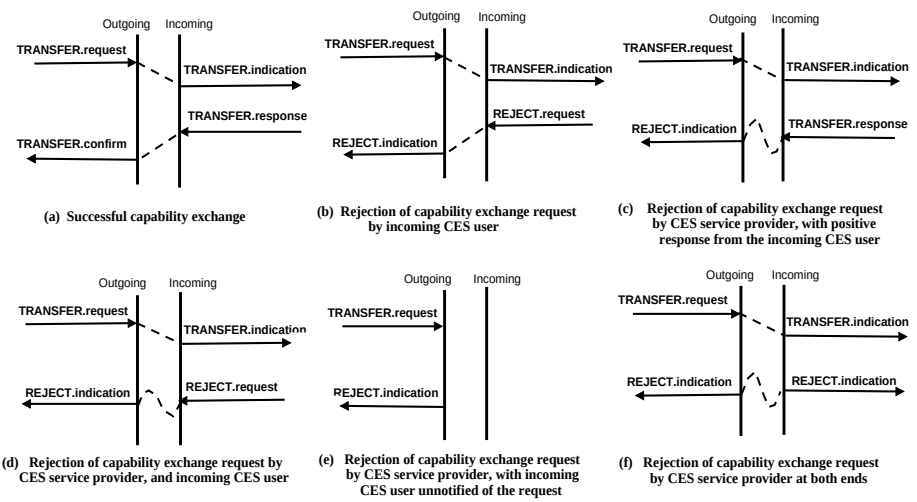


Fig. 3. Time sequence diagrams of the CES service for one transaction

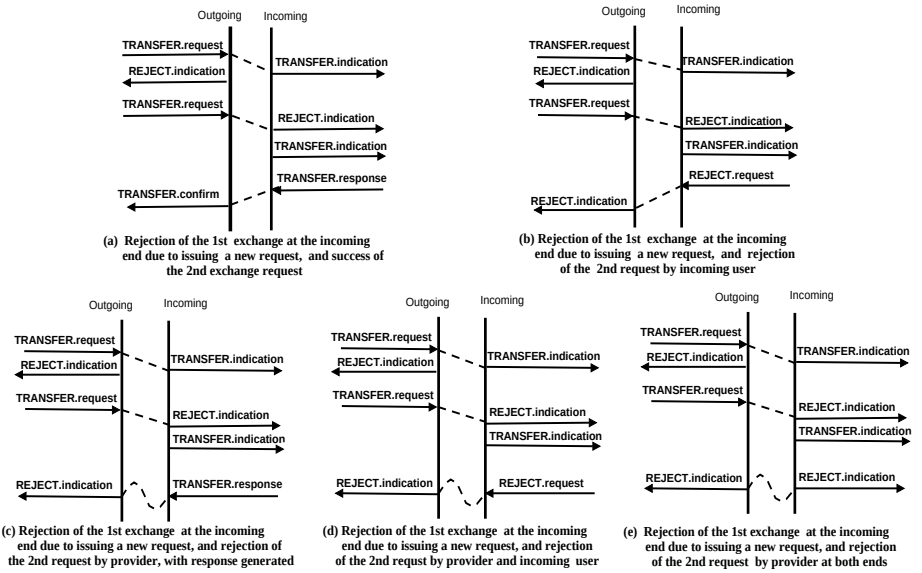


Fig. 4. Time sequence diagrams of the CES service for two transactions

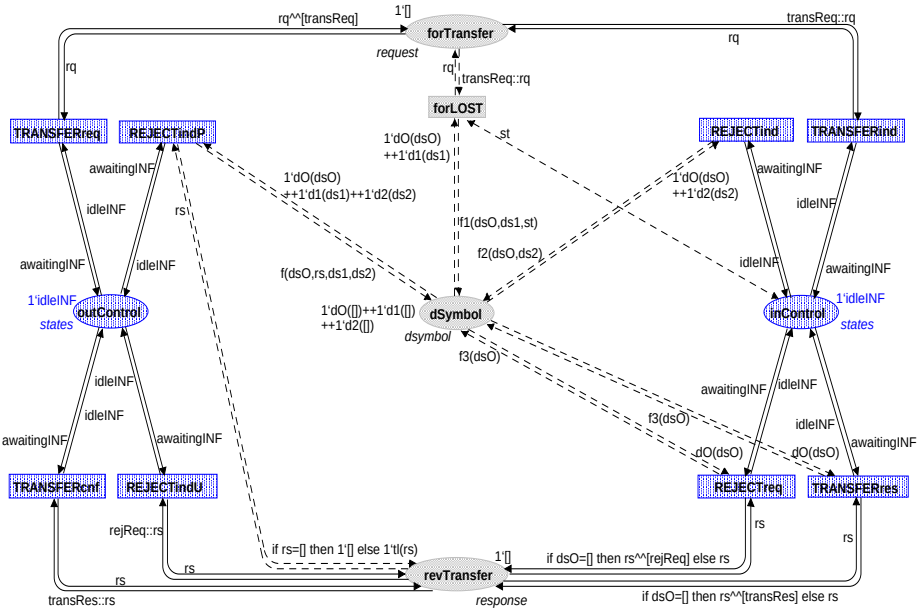
3 The CPN Model of the CES Service

Our primary goal of modelling the CES service with CPNs is to generate the CES service language. Hence the emphasis of modelling is on how to capture the relationships among the CES service primitives, and at this level of abstraction, the parameters of the CES service primitives are not modelled (except for the SOURCE parameter of primitive REJECT.indication at the outgoing end). Figure 5 depicts the CES service CPN model with the declarations at the bottom.

The darkly shaded places and transitions on the left side of the CPN correspond to the interface between the outgoing CES service user and the service provider in Fig. 1. This part of the CPN models the local sequences of primitives at the outgoing interface that is defined in Fig. 2(a). In the same way, the darkly shaded part on the right side of the CPN models the interface between the incoming user and the service provider.

In the middle of the CPN, the 3 lightly shaded places, transition and dashed arcs model the behaviour of the CES service provider. The service provider is modelled so that the end-to-end correlation of the primitives specified in Figs. 3 and 4 are captured, while the detailed operation of the service provider is hidden.

The following introduces this CPN model in more detail. We first look at how the local sequences of primitives are modelled, i.e. the CPN parts modelling the outgoing and incoming interfaces. Then we describe how the end-to-end correlation of the primitives is modelled, i.e. the middle part of the CPN representing the abstract behaviour of the CES service provider.




```
color states = with idleINF | awaitingINF;          color req = with transReq;
color request = list req;                           color res = with transRes | rejReq;
color response = list res;                          color dsym = with d;
color dsymb = list dsym;                            color dsymbol = union dO: dsymb + d1: dsymb + d2:
dsymb;

var rq: request;      var rs: response;             var dsO,ds1,ds2: dsymb;      var st: states;

fun f1(dsO,ds1,st) = if dsO=[] orelse (st=awaitingINF andalso length(dsO)=1)
   then 1'dO(dsO)++1'd1(ds1^^[d])
   else 1'dO(tl(dsO))++1'd1(ds1);
fun f2(dsO,ds2) = if dsO=[]  then 1'dO(dsO)++1'd2(ds2^^[d])
   else 1'dO(tl(dsO))++1'd2(ds2);
fun f3(dsO) = if dsO=[]   then 1'dO(dsO)
   else 1'dO(tl(dsO));
fun f(dsO,[],[],[])=1'dO(dsO^^[d])++1'd1([])++1'd2([])
  | f(dsO,[],[],d::ds2)=1'dO(dsO)++1'd1([])++1'd2(ds2)
  | f(dsO,[],d::ds1,[])=1'dO(dsO)++1'd1(ds1)++1'd2([])
  | f(dsO,r::rs,[],[])=1'dO(dsO)++1'd1([])++1'd2([])
  | f(_,_,_,_) = empty;
```

Fig. 5. The CES service CPN model

3.1 Modelling the Local Sequences of Primitives

The two darkly shaded parts on the left and right sides of the CPN encode the state transition diagrams in Fig. 2. They model the sequences of primitives at the outgoing and incoming interfaces accordingly.

The states of the two interfaces are modelled by places `outControl` and `inControl` respectively. They both are typed by "states" (see the declarations), which has two colours "idleINF" and "awaitingINF", corresponding to the two states "IDLE" and "AWAITING RESPONSE" of the interfaces.

The primitives are modelled as CPN transitions. For simplicity, all the primitive names are abbreviated. Their general names (i.e. TRANSFER and REJECT) are kept; the type names (i.e. request, indication, response and confirm)

are abbreviated to "req", "ind", "res" and "cnf" respectively; and the "." in each name is omitted. Moreover, at the outgoing side, the REJECT.indication primitive is modelled as two separate transitions `REJECTindP` and `REJECTindU`, corresponding to the different values of the SOURCE parameter of this primitive (Table 1). Here "P" indicates service provider initiated rejection (SOURCE = PROTOCOL) and "U" rejection by the incoming user (SOURCE = USER).

3.2 Modelling the End-to-End Correlation among Primitives

Places `forTransfer`, `revTransfer`, and `dSymbol`, transition `forLOST`, and associated arcs are used to capture the end-to-end correlation of the primitives.

Place `forTransfer` and the arcs connecting it and the transitions at the two ends model that the provider can pass the requests of the outgoing user to its peer. This place has the colour set "request", a list of "req". Variable "rq" has the type "request", which is used to concatenate and extract an element to/from the queue in place `forTransfer`. It may be argued that it is simpler to define the colour set of place `forTransfer` as "req" instead of a list of "req", but we use the list to facilitate the analysis in Section 4 and Section 5.

Place `revTransfer` and the related arcs model that the provider can transmit the responses of the incoming user to the outgoing user. It is typed by colour set "response", a list of "res". Colour set "res" has two colours, "transRes" and "rejReq", corresponding to the acceptance and rejection of the capability exchange request from the incoming user respectively.

Transition `forLOST` and the associated arcs model that the requests from the outgoing user may be lost by the service provider during transmission. The responses from the incoming user also can be lost, but we will get the same sequences of primitives as for the case of service provider initiated rejection shown in Figs. 3(c) and 3(d). So no extra transition is used to drop a response from place `revTransfer`. Instead, arcs are used between place `revTransfer` and transition `REJECTindP` to collect a "lost" response (the garbage) from this place when `REJECTindP` occurs.

While the purpose of using the above places and transition is obvious, the meaning for place `dSymbol` and its related arcs is relatively complex. We know that, as long as the outgoing interface is idle, the CES service allows a new TRANSFER.request primitive to be issued when there are outstanding capability exchanges. The newly issued TRANSFER.request invalidates these outstanding exchanges and the outgoing CES user always expects the response to the most recently issued request. However, for those outstanding exchanges, the responses may have been generated by the incoming side and put into place `revTransfer`. Then the service provider has to discard the invalid responses, otherwise, at the outgoing side, transitions `TRANSFERcnf` or `REJECTindU` can occur incorrectly, as a result of the invalid responses being in place `revTransfer`.

We could have identified each request and its response by including a sequence number, e.g. an integer (starting from 0), then in the model, it would be very easy to distinguish those unexpected responses since they must have different sequence numbers from that of the newly issued request. However, it is not

desirable to model a service at such a detailed level, and using sequence numbers may lead to state explosion. The key to correctly discarding the invalid responses is to coordinate the occurrence of transition `REJECTindP` at the outgoing side with transitions `REJECTind`, `REJECTreq` and `TRANSFERres` at the incoming side and transition `forLOST`.

If a capability exchange is successful or rejected by the incoming user (Figs. 3(a) and 3(b)), the response in `revTransfer` will be absorbed by the occurrence of `TRANSFERcnf` or `REJECTindU`. Then a new request can be issued by the outgoing user. For all other cases, transition `REJECTindP` must occur before a new request can be issued by the outgoing user. Correspondingly, one of the four transitions, `REJECTind`, `forLOST`, `REJECTreq`, and `TRANSFERres` has to occur. The problem, however, is the ordering of the occurrence of `REJECTindP` and each of the above four transitions.

We use place `dSymbol` and its related arcs to deal with the invalid responses when `REJECTindP` occurs. The basic idea is as follows. For a specific capability exchange, when `REJECTindP` occurs before one of those four transitions, a symbol is put into place `dSymbol` to control the output of the next occurrence of one of the four transitions. If `REJECTind` or `forLOST` occurs, no response is generated. If transition `REJECTreq` or `TRANSFERres` occurs, we do not allow any response to be put into `revTransfer`, modelling that the response is removed by the service provider. When transition `REJECTind` or `forLOST` occurs before `REJECTindP`, a different symbol is put into place `dSymbol` to indicate their occurrence. When transition `TRANSFERres` or `REJECTreq` occurs before `REJECTindP`, the corresponding response is put in place `revTransfer`. Then when `REJECTindP` occurs later, it cleans up either the symbol put in place `dSymbol` by the occurrence of `REJECTind` or `forLOST`, or the response in place `revTransfer`. This mechanism finally ensures that for every request issued by the outgoing side, if it is terminated at the outgoing side (i.e. `REJECTindP` occurs), its response (if any) is discarded by the service provider before a new request can be issued.

From the above, we also can see that, at any time, at most one response can be in place `revTransfer`. This design simplifies the state space of the CPN.

4 Validating the CES Service CPN Model

To validate the correctness of the CPN model, we first use Design/CPN [11] to interactively simulate this model and observe the possible occurrence sequences. During the simulation, we do not set any limitations to the capacity of the places in Fig. 5, hence any number of "transReq"s can stay in the queue of place `forTransfer`. The sequences we have observed are all consistent with the state transition diagrams (Fig. 2) and the time sequence diagrams (Figs. 3 and 4).

Since only a subset of all the possible sequences can be viewed by simulation, we also try to validate the model by generating its OG. We hope to observe all the possible occurrence sequences from the OG and to check whether they are all consistent with the specification in Figs. 2, 3 and 4.

However, even though we do not use sequence numbers, this model has an infinite state space. This is caused by the infinite occurrence of transition `TRANSFERreq` followed by transition `REJECTindP`, resulting in place `forTransfer` containing an infinite queue. We can limit the size of the state space by fixing the maximum queue length of place `forTransfer` (denoted as ML). Referring to the CPN in Fig. 5, we can see that limiting ML is equivalent to setting the capacity of the channel to a specific value. This does not prevent us from checking that the CPN models the correct primitive correlations. Therefore, to start with, we let $ML = 1$. This is done by simply adding a guard, $[length(rq) < 1]$ to transition `TRANSFERreq`, which ensures that only when place `forTransfer` has an empty list, a "transReq" can be put into it.

We obtain the OG of Fig. 6 for this case. At the bottom right of this figure are the mnemonics for the transitions of the CPN model and the same symbols will be used in the OGs in Section 5. We have examined the occurrence sequences of transitions included in this OG and have not found any sequences which are inconsistent with those defined in Figs. 2, 3 and 4. Also all the sequences defined in these figures are included in this OG. This raises our confidence in the correctness of the CPN model. We now turn our attention to dealing with the infinite state space of the CPN model.

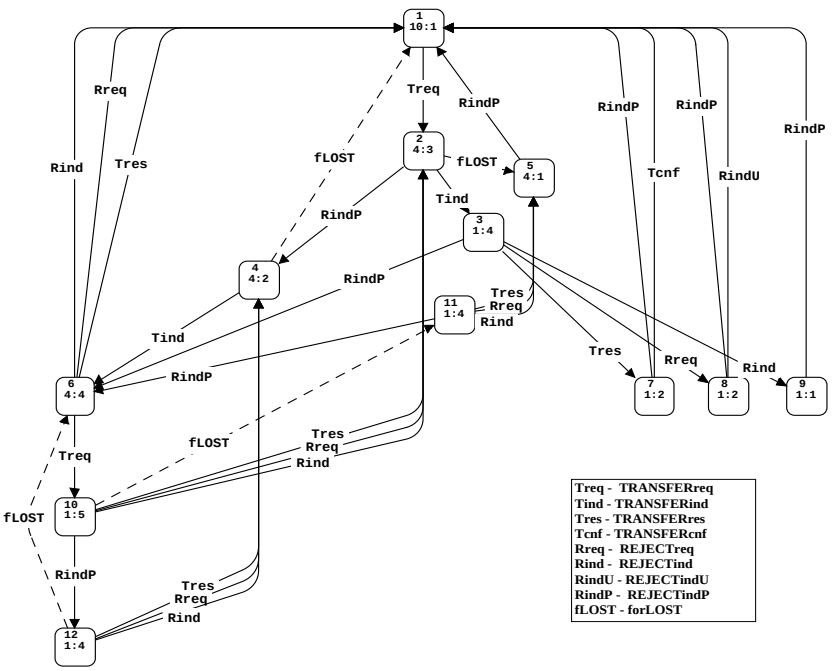


Fig. 6. The *OG* of the CES service CPN model when $ML = 1$

5 Tackling the Infinite State Space of the CES Service CPN Model

The key to obtaining a tractable state space for any given value of the maximum queue length (ML) is to examine its structure as ML is varied. Figure 7 shows the OG when $ML = 2$. We find that the OG for $ML = 2$ can be obtained from the OG for $ML = 1$, by adding 4 nodes and 16 arcs (the shaded nodes and bold arcs). Moreover, the added part is connected to four nodes of the OG when $ML = 1$ (i.e. nodes 2, 4, 10 and 12 in Figs. 6 and 7).

On generating the OG when $ML = 3$ (Fig. 8), we find that it has a similar relationship with the OG for $ML = 2$. Moreover, the same is true when generating the OGs when $ML = 4, 5, \ldots, 10$. These results give us the intuition that the size of the OG (number of nodes and arcs) is linear in ML. More importantly, the structure of the OGs suggests that we can construct the OG for any value of ML, say l, from the OG for $ML = l - 1$.

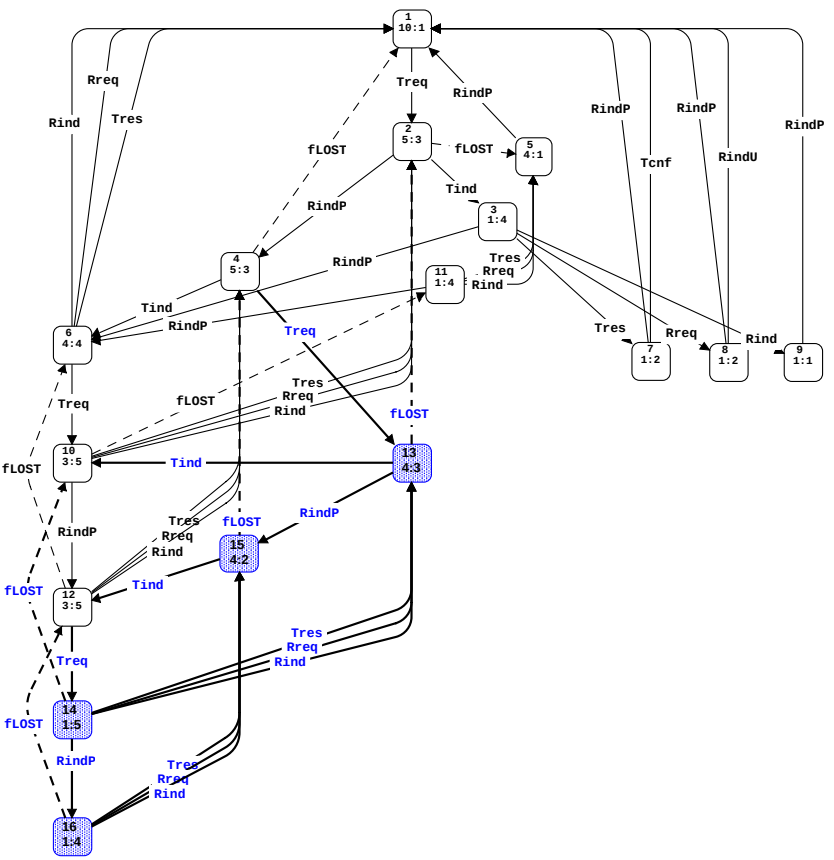


Fig. 7. The *OG* of the CES service CPN model when $ML = 2$

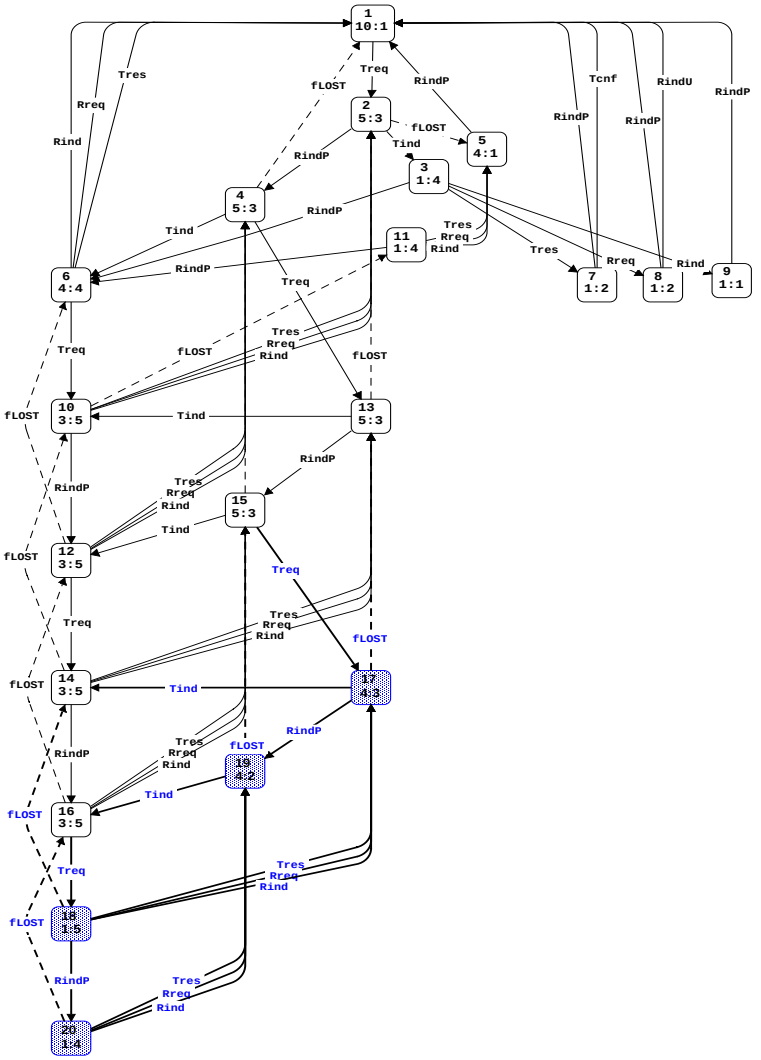


Fig. 8. The *OG* of the CES service CPN model when $ML = 3$

5.1 A Recursive Formula for the OG

We propose a recursive formula for the OG and prove it in this section. Firstly, we define the OG of a CPN based on the definition in [15].

Definition 1 *The* **OG** *of a CPN is a labelled directed graph* $OG = (V, A)$ *where*

(i) $V = [M_0\rangle$*, is a set of nodes.*
(ii) $A = \{(M_1, b, M_2) \in V \times BE \times V \mid M_1[b\rangle M_2\}$*, is a set of labelled arcs,* BE *is the set of binding elements of the CPN, and* $M_1[b\rangle M_2$ *denotes that the*

marking of the CPN changes from M_1 to M_2 on the occurrence of binding element $b \in BE$.

Then we parameterize the CES service CPN model with the value of ML, and have the following definition.

Definition 2 *For $l \in \mathcal{N}^+$ (the positive integers), CPN_l is defined as the CES service CPN model of Fig. 5, augmented with the guard $[length(rq) < l]$ for transition* `TRANSFERreq`.

Next we give the theorem and prove it. Firstly, let OG_l be the occurrence graph for CPN_l.

Theorem 1 *For $l \in \mathcal{N}^+$, $OG_l = (V_l, A_l)$ is given by:*

(i) $OG_1 = (V_1, A_1)$ is the OG of Fig. 6.

(ii) for $l \geq 2$, $OG_l = (V_{l-1} \cup V_l^{add}, A_{l-1} \cup A_l^{add})$ where
- *V_l^{add} is a subset of V_l, shown in Table 2; and*
- *$A_l^{add} = \{(M_i, b, M_j) \in (V_{l-1}^{add} \times BE \times V_l^{add} \cup V_l^{add} \times BE \times V_{l-1}^{add}) \mid M_i[b\rangle M_j\}$, given in Table 3.*

Table 2 shows the details of the set of nodes required to obtain OG_l from OG_{l-1}. Their names are shown in the first column. The subscript (l) of the names indicates that they are nodes of the OG when $ML = l$, and the superscripts (add_1 to add_4) represent that they are the added nodes with respect to OG_{l-1}. The remaining 5 columns of this table show the markings of each place in Fig. 5 for these nodes. We can see that each node has l "transReq"s in place "forTransfer" and no response in place "revTransfer". In Table 3, the names of the added arcs defined in **(ii)** are shown in the first column, and the subscripts and superscripts have the same meaning as those in Table 2. The arcs are described as triples in the second column, where the first element is the source node, the second element is the label (i.e. a binding element), and the last element is the destination node of the arc.

Before the proof of **Theorem 1**, as an example, we look at how OG_3 (Fig. 8) can be represented by OG_2 (Fig. 7) and a set of added nodes and arcs with respect to OG_2. If we ignore the shaded nodes and their associated arcs (i.e. the bold arcs) in Fig. 8, we can see that the remaining part is the same as the OG_2 in Fig. 7. So OG_2 is a subgraph of OG_3. By observing the markings of nodes 13, 14, 15 and 16 generated using Design/CPN (Fig. 9), we find that they can be obtained by substituting 2 for l in Table 2. So these four nodes make up V_2^{add}. Similarly, we find that V_3^{add} consists of nodes 17, 18, 19 and 20 in Fig. 8, because the markings of these nodes (Fig. 9) can be obtained by substituting 3 for l in Table 2. Finally, A_3^{add} are the bold arcs in Fig. 8 since they are the same arcs in Table 3 when substituting 3 for l. Thus we have verified that OG_3 can be represented as a combination of OG_2 and a set of added nodes and arcs according to **Theorem 1**.

Table 2. V_l^{add} $(l \in \mathcal{N}^+)$

Node	outControl	inControl	forTransfer	revTransfer	dSymbol
$M_l^{add_1}$	1`awaitingINF	1`idleINF	$1`[\underbrace{transReq, \cdots, transReq}_{l}]$	1`[]	$1`dO(\underbrace{[d, \cdots, d]}_{l-1})++1`d1([])++1`d2([])$
$M_l^{add_2}$	1`awaitingINF	1`awaitingINF	$1`[\underbrace{transReq, \cdots, transReq}_{l}]$	1`[]	$1`dO(\underbrace{[d, \cdots, d]}_{l})++1`d1([])++1`d2([])$
$M_l^{add_3}$	1`idleINF	1`idleINF	$1`[\underbrace{transReq, \cdots, transReq}_{l}]$	1`[]	$1`dO(\underbrace{[d, \cdots, d]}_{l})++1`d1([])++1`d2([])$
$M_l^{add_4}$	1`idleINF	1`awaitingINF	$1`[\underbrace{transReq, \cdots, transReq}_{l}]$	1`[]	$1`dO(\underbrace{[d, \cdots, d]}_{l+1})++1`d1([])++1`d2([])$

In the following, we prove **Theorem 1** by induction over the maximum queue length l. Firstly we prove **(i)** of the theorem, then given that **(ii)** holds for l, we prove that it holds for $l+1$.

Proof. We assume that Design/CPN can generate the correct OG of a CPN model with a finite state space, then **(i)** is proved.

Suppose that, based on OG_{l-1}, OG_l can be obtained through **(ii)**. We now prove that **(ii)** is true for OG_{l+1}. The proof is divide into two parts.

Firstly, we prove that if we generate a subgraph of OG_{l+1} by only considering markings that are generated under the condition that $ML \leq l$, then this subgraph is the same as OG_l.

Secondly, we prove that the additional nodes and arcs of OG_{l+1}, when $ML = l+1$ are those given in Tables 2 and 3, when substituting $l+1$ for l.

We firstly create CPN_l, according to **Definition 2**, which is the CPN of Fig. 5 with the addition that transition `TRANSFERreq` has a guard $[length(rq) < l]$. When we create CPN_{l+1} by substituting $l+1$ for l in the guard, the only

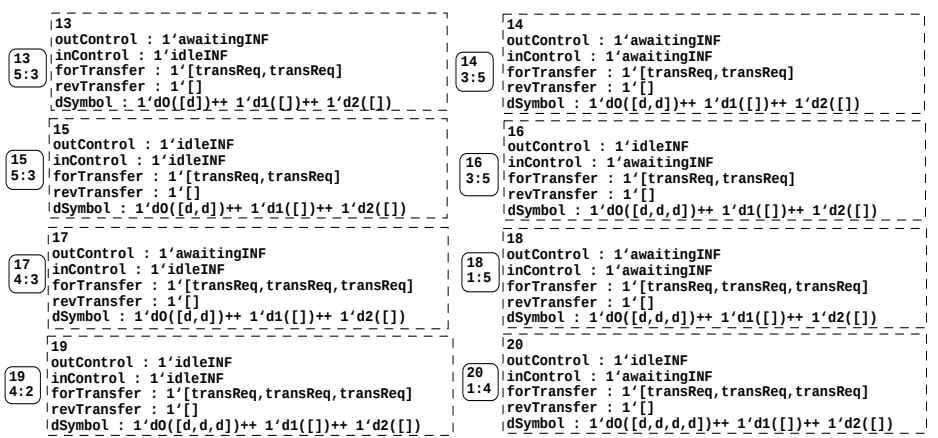


Fig. 9. Markings of nodes 13 to 20 in Fig. 8

Table 3. A_l^{add} $(l \in \mathcal{N}^+)$

Name	Arc
$a_l^{add_1}$	$(M_{l-1}^{add_3}, (Treq, < rq = [\underbrace{transReq, \cdots, transReq}_{l-1}] >), M_l^{add_1})$
$a_l^{add_2}$	$(M_{l-1}^{add_4}, (Treq, < rq = [\underbrace{transReq, \cdots, transReq}_{l-1}] >), M_l^{add_2})$
$a_l^{add_3}$	$(M_l^{add_1}, (RindP, < dsO = [\underbrace{d, \cdots, d}_{l-1}], ds1 = [], ds2 = [], rs = [] >), M_l^{add_3})$
$a_l^{add_4}$	$(M_l^{add_1}, (Tind, < rq = [\underbrace{transReq, \cdots, transReq}_{l}] >), M_{l-1}^{add_2})$
$a_l^{add_5}$	$(M_l^{add_1}, (fLOST, < rq = [\underbrace{transReq, \cdots, transReq}_{l}], st = idleINF, dsO = [\underbrace{d, \cdots, d}_{l-1}], ds1 = [] >), M_{l-1}^{add_1})$
$a_l^{add_6}$	$(M_l^{add_2}, (RindP, < dsO = [\underbrace{d, \cdots, d}_{l}], ds1 = [], ds2 = [], rs = [] >), M_l^{add_4})$
$a_l^{add_7}$	$(M_l^{add_2}, (fLOST, < rq = [\underbrace{transReq, \cdots, transReq}_{l}], st = awaitingINF, dsO = [\underbrace{d, \cdots, d}_{l}], ds1 = [] >), M_{l-1}^{add_2})$
$a_l^{add_8}$	$(M_l^{add_2}, (Tres, < dsO = [\underbrace{d, \cdots, d}_{l}], rs = [] >), M_l^{add_1})$
$a_l^{add_9}$	$(M_l^{add_2}, (Rreq, < dsO = [\underbrace{d, \cdots, d}_{l}], rs = [] >), M_l^{add_1})$
$a_l^{add_{10}}$	$(M_l^{add_2}, (Rind, < dsO = [\underbrace{d, \cdots, d}_{l}], ds2 = [] >), M_l^{add_1})$
$a_l^{add_{11}}$	$(M_l^{add_3}, (Tind, < rq = [\underbrace{transReq, \cdots, transReq}_{l}] >), M_{l-1}^{add_4})$
$a_l^{add_{12}}$	$(M_l^{add_3}, (fLOST, < rq = [\underbrace{transReq, \cdots, transReq}_{l}], st = idleINF, dsO = [\underbrace{d, \cdots, d}_{l}], ds1 = [] >), M_{l-1}^{add_3})$
$a_l^{add_{13}}$	$(M_l^{add_4}, (Tres, < dsO = [\underbrace{d, \cdots, d}_{l+1}], rs = [] >), M_l^{add_3})$
$a_l^{add_{14}}$	$(M_l^{add_4}, (Rreq, < dsO = [\underbrace{d, \cdots, d}_{l+1}], rs = [] >), M_l^{add_3})$
$a_l^{add_{15}}$	$(M_l^{add_4}, (Rind, < dsO = [\underbrace{d, \cdots, d}_{l+1}], ds2 = [] >), M_l^{add_3})$
$a_l^{add_{16}}$	$(M_l^{add_4}, (fLOST, < rq = [\underbrace{transReq, \cdots, transReq}_{l}], st = awaitingINF, dsO = [\underbrace{d, \cdots, d}_{l+1}], ds1 = [] >), M_{l-1}^{add_4})$

difference is the enabling of transition `TRANSFERreq`. Suppose we start to construct OG_{l+1} (for CPN_{l+1}) from the initial marking. Firstly, we only generate nodes with l or less "transReq"s in place `forTransfer`. We obtain a subgraph of OG_{l+1} that includes all the markings of V_{l+1} with l or less "transReq"s in place `forTransfer`, and the associated arcs. By the definition of OG_l, we know that this subgraph of OG_{l+1} is $OG_l = (V_l, A_l)$.

Next, nodes with $l+1$ "transReq"s in place `forTransfer`, and the associated arcs (i.e. V_{l+1}^{add} and A_{l+1}^{add}) are generated. From Fig. 5, the only way of obtaining $l+1$ "transReq"s in `forTransfer` is for there to be l "transReq"s in place `forTransfer` and for `TRANSFERreq` to occur. Therefore predecessors of nodes in V_{l+1}^{add} must have l "transReq"s in place `forTransfer` and an "idleINF" in place `outControl`.

In the above, we have proved that OG_l is the subgraph of OG_{l+1} which includes all the nodes of OG_{l+1} with l "transReq"s in place `forTransfer`, so the predecessors must be in the subgraph, i.e. OG_l. Moreover, because we assume that **(ii)** holds for OG_l, V_l^{add} (Table 2) includes all the nodes of OG_l with l "transReq"s in place `forTransfer`. According to Table 2, only nodes $M_l^{add_3}$ and $M_l^{add_4}$ have $M(outControl) = 1`idleINF$, so they are the only predecessors. In the following, we construct V_{l+1}^{add} and A_{l+1}^{add} starting from these two nodes. When referring to Tables 2 and 3 in the rest of the proof, if not pointed out specifically, the l in each table is substituted by $l+1$.

At node $M_l^{add_3}$, transitions `TRANSFERreq`, `TRANSFERind` and `forLOST` are enabled. However, only the occurrence of `TRANSFERreq` can put the $(l+1)$th "transReq" into place `forTransfer`, creating a node of V_{l+1}^{add}. When it occurs, we have $M_{l+1}^{add_1}$ (Table 2), and the arc from $M_l^{add_3}$ to it, $a_{l+1}^{add_1} \in A_{l+1}^{add}$ (Table 3).

At $M_l^{add_4}$, the enabled transitions are `TRANSFERres`, `REJECTreq`, `REJECTind`, `forLOST`, and `TRANSFERreq`. Similarly, only the occurrence of `TRANSFERreq` can generated a node of V_{l+1}^{add}, which is $M_{l+1}^{add_2}$ (Table 2), and the arc from $M_l^{add_4}$ to $M_{l+1}^{add_2}$, $a_{l+1}^{add_2}$ (Table 3) is in A_{l+1}^{add}.

Now we need to check, from the generated nodes, $M_{l+1}^{add_1}$ and $M_{l+1}^{add_2}$, if any more nodes and arcs of V_{l+1}^{add} and A_{l+1}^{add} can be constructed.

At $M_{l+1}^{add_1}$, transitions `REJECTindP`, `TRANSFERind` and `forLOST` are enabled. When `REJECTindP` occurs, we obtain $M_{l+1}^{add_3} \in V_{l+1}^{add}$ (Table 2), and the arc from $M_{l+1}^{add_1}$ to it, $a_{l+1}^{add_3} \in A_{l+1}^{add}$ (Table 3). The occurrence of `TRANSFERind` or `forLOST` takes one "transReq" from place `forTransfer`, so the destination node for each case is not in V_{l+1}^{add}, but the arcs from $M_{l+1}^{add_1}$ to the two destination nodes are in A_{l+1}^{add}, which are $a_{l+1}^{add_4}$ and $a_{l+1}^{add_5}$ respectively (Table 3).

At node $M_{l+1}^{add_2}$, transitions `REJECTindP`, `forLOST`, `TRANSFERres`, `REJECTreq` and `REJECTind` are enabled. When `REJECTindP` occurs, we obtain $M_{l+1}^{add_4} \in V_{l+1}^{add}$ (Table 2) and arc $a_{l+1}^{add_6} \in A_{l+1}^{add}$ (Table 3). When `forLOST` occurs, the state of the CPN changes to $M_l^{add_2}$ (Table 2 without substituting $l+1$ for l), which is a node of OG_l. However, the arc $a_{l+1}^{add_7}$ (Table 3) from $M_{l+1}^{add_2}$ to $M_l^{add_2}$ is in A_{l+1}^{add}. When any of the last three transitions occurs, the state of the CPN transits to the same node, $M_{l+1}^{add_1}$ (Table 2), and we obtain arcs $a_{l+1}^{add_8}$, $a_{l+1}^{add_9}$ and $a_{l+1}^{add_{10}}$ (Table 3), which are all in A_{l+1}^{add}.

Therefore from nodes $M_{l+1}^{add_1}$ and $M_{l+1}^{add_2}$, we construct two more nodes of V_{l+1}^{add}, $M_{l+1}^{add_3}$ and $M_{l+1}^{add_4}$. We now need to investigate the possible nodes and arcs of V_{l+1}^{add} and A_{l+1}^{add} which can be generated from them.

At $M_{l+1}^{add_3}$, transition `TRANSFERind` or `forLOST` can occur. The occurrences of either of them results in a node with l "transReq"s in place `forTransfer`, hence no new node of V_{l+1}^{add} is generated. However, the arcs from $M_{l+1}^{add_3}$ to the two resulting nodes, i.e. $a_{l+1}^{add_{11}}$ and $a_{l+1}^{add_{12}}$ (Table 3) are in A_{l+1}^{add}.

At $M_{l+1}^{add_4}$, transitions `TRANSFERres`, `REJECTreq`, `REJECTind` and `forLOST` are enabled. When any of the first three transitions occurs, the CPN state goes

to the same node, $M_{l+1}^{add_3}$, and we obtain arcs $a_{l+1}^{add_{13}}$, $a_{l+1}^{add_{14}}$, $a_{l+1}^{add_{15}}$ respectively (Table 3), and they are in A_{l+1}^{add}. When `forLOST` occurs, a node with l "transReq"s in place `forTransfer` is reached via arc $a_{l+1}^{add_{16}}$ (Table 3). While this node is not in V_{l+1}^{add}, $a_{l+1}^{add_{16}}$ is in A_{l+1}^{add}.

Thus, we have constructed all the directly reachable markings from nodes $M_l^{add_3}$ and $M_l^{add_4}$ which are in V_{l+1}^{add}, and all of their successors which are in V_{l+1}^{add}, as well as the related arcs. Altogether, we obtain 4 nodes for V_{l+1}^{add}, and 16 arcs for A_{l+1}^{add}. Moreover, V_{l+1}^{add} and A_{l+1}^{add} satisfy Tables 2 and 3 respectively for OG_{l+1}. So we have proved that OG_{l+1} can be obtained by combining OG_l, V_{l+1}^{add} and A_{l+1}^{add}, for which Tables 2 and 3 are true. Therefore **(ii)** holds.

Combining the proofs of **(i)** and **(ii)**, the theorem is proved. □

5.2 Formulae for the Size of the OG

From the proof of **Theorem 1**, it can be seen that the size of the state space for $l \in \mathcal{N}^+$ is linear in l, and is given by

$$|V_l| = 12 + 4(l-1), \quad and \quad |A_l| = 33 + 16(l-1)$$

5.3 An Algorithm for Recursively Generating the OG

In Section 5.1, we propose and prove a recursive formula for the OG of the parameterized CES service CPN CPN_l. From the proof of **Theorem 1**, we also obtain the following algorithm to construct OG_l given OG_{l-1} ($l \geq 2$).

Algorithm 1 *Given the OG of CPN_{l-1}, $OG_{l-1} = (V_{l-1}, A_{l-1})$ ($l \geq 2$), OG_l can be constructed by applying the algorithm below.*

(i) Find out the two nodes $M_{l-1}^{add_3}, M_{l-1}^{add_4} \in V_{l-1}^{add}$ (Table 2 with substitution of $l-1$ for l).
(ii) From $M_{l-1}^{add_3}$, add arc $a_l^{add_1}$ (Table 3) and its destination node $M_l^{add_1}$ (Table2).
(iii) From $M_{l-1}^{add_4}$, add arc $a_l^{add_2}$ (Table 3) and its destination node $M_l^{add_2}$ (Table 2).
(iv) From $M_l^{add_1}$, add arc $a_l^{add_3}$ (Table 3) and its destination node $M_l^{add_3}$ (Table 2).
(v) From $M_l^{add_1}$, add arcs $a_l^{add_4}$ and $a_l^{add_5}$ (Table 3).
(vi) From $M_l^{add_2}$, add arcs $a_l^{add_6}$ and its destination node $M_l^{add_4}$ (Table 2).
(vii) From $M_l^{add_2}$, add arcs $a_l^{add_7}$, $a_l^{add_8}$, $a_l^{add_9}$ and $a_l^{add_{10}}$ (Table 3).
(viii) From $M_l^{add_3}$, add arcs $a_l^{add_{11}}$ and $a_l^{add_{12}}$ (Table 3).
(ix) From $M_l^{add_4}$, add arcs $a_l^{add_{13}}$, $a_l^{add_{14}}$, $a_l^{add_{15}}$ and $a_l^{add_{16}}$ (Table 3).

The OG obtained is OG_l.

6 Conclusions and Future Work

The results presented in this paper are motivated by our desire to verify the Capability Exchange Signalling protocol definition of H.245 [14]. Our previous work [17,18] concentrated on modelling and analysing the CES protocol. To verify the protocol, we need to specify the service it is to provide to its users, and then to check that the protocol does provide this service. Our methodology for verifying the protocol against its service is based on the notion of *service language* [6], which defines all the sequences of service events (primitives) and can be generated from the OG of the CPN model of the service.

Because the CES service is not completely defined in H.245, our first contribution is the complete CES service definition given in Section 2. Secondly we obtain a general CPN model of the CES service. While this model is designed for generating the CES service language, it also provides an abstract formal specification of the CES service, which is new. Because the OG of this CPN model is infinite, generating the CES service language from the CPN model is not straightforward.

This paper has focussed on tackling the infinite state space of the CES service CPN. We parameterize the CPN by introducing a communication channel capacity. We then develop a recursive formula for the OG of the parameterized CPN, prove it to be correct and state an algorithm to construct the OG. The algorithm allows the OG to be developed incrementally, instead of from the beginning for each capacity of the channel. Further we provide formulae for the size of the state space (nodes and arcs) that show it is linear in the channel capacity.

Future work will involve investigating the service language using automata reduction techniques to eliminate the internal `forLOST` transition, while preserving the sequences of primitives. We shall investigate if a similar recursive formula can be used to describe the reduced automaton and thus the CES service language for any capacity of the communication channel. If we succeed, we will apply it to verify the CES protocol against its service specification for realistic communication channel capacities.

We are also considering generalising the result of this case study to other applications. One possible application is to represent state spaces of communication *protocols* that have multiple transactions and operate over a transport medium with unbounded or very high capacity. Such a state space may also be represented recursively in terms of the capacity of the medium. An example of these protocols is the CES protocol itself. When analysing the properties of the CES protocol in [18], we limit the capacity of the transport medium to a small capacity. In the future, we shall try to obtain a general expression of its state space for arbitrary capacity. Once this is achieved, we can use it to prove properties of the CES protocol for *any* given capacity of its underlying transport medium. Furthermore, if the recursive formula of the CES service language can be obtained, we hope to be able to verify the CES protocol against its service for any communication channel capacity as well. We may also use the approach of this paper to study state spaces that have a repetitive structure, such as that of the distributed missile simulator presented in [9].

References

1. Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels. In *Proc. of the 10th Int. Conf. on Computer Aided Verification (CAV'98)*, pages 305–318, 1998. 274
2. Parosh Aziz Abdulla and Bengt Jonsson. Ensuring completeness of symbolic verification methods for infinite-state systems. *Theoretical Computer Science*, 256:145–167, 2001. 274
3. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *Proc. of the 13th Int. Conf. on Computer Aided Verification (CAV'01)*, Paris, Jul. 2001. Springer-Verlag. 274
4. W. A. Barret and J. D. Couch *Compiler Construction: Theory and Practice*. Science Research Associates, 1979. 274
5. Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Verification of Parameterized Protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001. 274
6. J. Billington, M. C. Wilbur-Ham, and M. Y. Bearman. Automated Protocol Verification. In M. Diaz, editor, *Protocol Specification, Testing and Verification, V*, pages 59–70. Elsevier Science Publisher B. V. (North-Holland), 1986. 274, 291
7. Jonathan Billington. Formal Specification of Protocols: Protocol Engineering. In Allen Kent, James G. Williams, and Rosalind Kent, editors, *Encyclopedia of Microcomputers, Vol. 7*, pages 299–314. Marcel Dekker, Inc.,1991. 274
8. Jonathan Billington, Michel Diaz, and Grzegorz Rozenberg, editors. *Application of Petri Nets to Communication Networks: Advances in Petri Nets*. LNCS 1605. Springer, 1999. 274
9. Steven Gordon and Jonathan Billington. Analysing a Missile Simulator with Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2), Dec. 1998. 291
10. Steven Gordon and Jonathan Billington. Modelling the WAP Transaction Service using Coloured Petri Nets. In *Proc. of the 1st Int. Conf. on Mobile Data Access* LNCS 1748, pages 109–118. Springer-Verlag, Dec. 1999. 274
11. Design/CPN homepage. http://www.daimi.au.dk/designCPN/. 275, 282
12. ITU-T. *ITU-T Recommendation X.210, Information Technology - Open Systems Interconnection - Basic Reference Model: Conventions for the Definition of OSI Services*, Nov. 1993. 274, 275, 276
13. ITU-T. *ITU-T Recommendation X.214, Information Technology - Open Systems Interconnection - Transport Service Definition*, Nov. 1995. 275
14. ITU-T. *ITU-T Recommendation H.245, Control Protocol for Multimedia Communication*, Nov. 2000. 273, 276, 291
15. Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, vol 2 and vol 3. Springer, 2nd edition, 1997. 274, 285
16. Markus Lindqvist. *Parameterized Reachability Trees for Predicate/Transition nets*, pages 301–324. LNCS 674. Springer-Verlag, 1993. 274
17. Lin Liu and Jonathan Billington. Modelling and Analysis of Internet Multimedia Protocols - Methodology and Initial Results. In *Proc. of the 11th Annual Int. Symp. of the Int. Council on Systems Engineering (INCOSE'2001)*, CD-ROM, paper 3.2.4, Melbourne, Australia, Jul. 2001. 274, 291
18. Lin Liu and Jonathan Billington. Modelling and Analysis of the CES Protocol of H.245. In *Proc. of the 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 95–114, Aarhus, Denmark, Aug. 2001. 274, 291

19. Andrew S. Tanenbaum. *Communication Networks*. Prentice-Hall International, Inc., third edition, 1996. 274
20. Antti Valmari and Ilkka Kokkarinen. Unbounded Verification Results by Finite-State Compositional Techniques: 10^{any} States and Beyond. In *Proc. of 1998 Int. Conf. on Application of Concurrency to System Design*, pages 75–85, Aizu-Wakamatsu, Fukushima, Japan, Mar. 1998. IEEE Computer Society. 274
21. Isabelle Vernier. Symbolic Excecutions of Symmetrical Parallel Programs. In *Proc. of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP'96)*, pages 327–324, Los Alamitos, CA, USA, 1996. IEEE Computer Society. 274

Modelling of Features and Feature Interactions in Nokia Mobile Phones Using Coloured Petri Nets

Louise Lorentsen[1], Antti-Pekka Tuovinen[2], and Jianli Xu[2]

[1] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
louisel@daimi.au.dk
[2] Software Technology Laboratory, Nokia Research Centre
P.O. Box 407, FIN-00045 Nokia Group, Finland
{antti-pekka.tuovinen,jianli.xu}@nokia.com

Abstract. This paper reports on the main results from an industrial cooperation project[1]. The project is a joint project between Nokia Research Centre and the CPN group at the University of Aarhus. The purpose of the project was to investigate features and feature interactions in development of Nokia mobile phones through construction of a Coloured Petri Nets (CPN) model. The model is extended with domain-specific graphics and Message Sequence Charts to enable mobile phone user interface designers and software developers who are not familiar with Petri Nets to work with the model. The paper presents the CPN model constructed in the project, describes how domain-specific graphics and Message Sequence Charts are used in simulations of the CPN model, and discusses how the project and in particular the construction of the CPN model has influenced the development process of features in Nokia mobile phones.

1 Introduction

Modern mobile phones provide an increasingly complex and diverse set of features. Besides basic communication facilities there is a growing demand for facilities for personal information management, data transmission, entertainment, etc. To support flexibility and smooth operation the user interface (UI) of the mobile phone is designed to support the most frequent and important user tasks, hence enabling many features to interplay and be active at the same time. The dependencies or interplay of features are called *feature interactions* and range from simple usage dependencies to more complex combinations of several independent behaviours.

When the project started, feature interactions were not systematically documented at Nokia and often the most complex feature interactions were not fully understood before the features were implemented. In the design and development of features, focus was often on the behaviour and appearance of individual

[1] The project is funded by Nokia Mobile Phones.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 294–313, 2002.
© Springer-Verlag Berlin Heidelberg 2002

features. The description of feature interactions was integrated in the description of the individual features involved and did not fully cover or treat feature interactions in a systematic way. This could, in the worst case, lead to costly delays in the integration phase of a set of independently developed features. Therefore, a need for more focus on the feature interactions in the development of features was identified.

The above mentioned problems have motivated the MAFIA[2] project. The main goals of the project were:

1. To increase the level of understanding of the role that feature interactions play in the UI software and its development by identification and documentation of typical patterns of feature interactions.
2. To develop a systematic methodology for describing feature interactions.
3. To provide an environment for interactive exploration and simulation of the feature interactions for demonstrational or analytical purposes.

One of the main activities in the MAFIA project was to construct a model using Coloured Petri Nets (CP-nets or CPNs) [14,17]. The use of CP-nets allowed us to have both a static graphical description of features and furthermore allowed simulation of the models and hence provided an environment for interactive exploration and simulation of feature interactions. CP-nets have been used in previous projects within Nokia, e.g., [23], and were therefore known to the project members in Nokia Research Centre.

The paper is structured as follows. Section 2 presents the MAFIA project and its organisation. Section 3 describes the CPN model constructed in the project. Section 4 discusses how the CPN model has been extended with domain-specific graphics and Message Sequence Charts. Section 5 contains a discussion of related and future work. Finally, Sect. 6 contains the conclusions and a discussion of how the construction of the CPN model has influenced the development of features in Nokia mobile phones. The reader is assumed to be familiar with the basic ideas of High-level Petri Nets [11].

2 The MAFIA Project

The MAFIA project was a joint project between Nokia Research Centre and the CPN group at the University of Aarhus. The project ran from November 2000 to November 2001 with a total amount of 15 man months of work resources. The project group consisted of two people from Nokia Research Centre and three people from the CPN group. Hence, the project group consisted of both experts from the application domain, i.e., development of mobile phones, and experts in the tools and techniques to be applied, i.e., CP-nets and its supporting tool Design/CPN [10].

[2] MAFIA is an acronym for Modelling and Analysis of Feature Interaction patterns in mobile phone software Architectures.

Before the joint project started, initial work was done at Nokia Research Centre to experiment with the use of CP-nets for the modelling of feature interactions. The researchers at Nokia Research Centre had practical experience with CP-nets and the Design/CPN tool from other Nokia research projects, e.g., [23]. Hence, the modelling work started immediately in the beginning of the project. When the joint project started, one researcher from the CPN group worked full time at Nokia Research Centre for six months. Other project members from the CPN group provided guidance and technical support on the modelling work.

In the first phase of the project overview information and internal documentation about the features and the mobile phone UI architecture were provided by Nokia and the project group experimented with different levels of abstraction in the CPN model. During the construction of the CPN model, the model was presented and discussed with both UI designers and software developers who were not familiar with Petri Nets. Therefore, already in the first phase of the project the project group decided to experiment with ideas for extending the CPN model with a layer of domain-specific graphics and Message Sequence Charts (MSCs) [13]. This was envisioned to enable the UI designers and software developers to work with the CPN model without interacting with the underlying CP-nets. Interactive simulations were used to validate the models. After the first phase of the project, the CPN modelling proceeded in three iterative steps each containing the following activities.

Refinement and Extension. The CPN model was refined and more features were added.

Validation. The CPN model was validated by means of both interactive and more automatic simulations.

Improvement of Visualisation Facilities. Both the domain-specific graphics and the use of Message Sequence Charts were improved and extended.

Presentation. During meetings with UI designers and software developers the CPN model was presented and its intended use was discussed.

Correction. Based on the evaluation in the meetings with the UI designers and software developers, the CPN model was corrected.

Hence, the CPN modelling proceeded with developing more and more elaborated models, each presented to the UI designers and software developers, i.e., the intended users of the CPN model. At the end of the project the CPN model and documentation were handed over to the users as a part of the results from the project. In Sect. 6 we will discuss how the CPN model relates to the other results from the MAFIA project and how the construction of the CPN model has influenced the development of features in Nokia.

3 The CPN Model

This section presents parts of the CPN model. The CPN model does not capture the full mobile phone UI software system but concentrates on a number

of selected features that are interesting seen from a feature interaction perspective. The purpose of this section is twofold. Firstly, to give an overview of the CPN model and secondly, to give an idea of the complexity and the abstraction level chosen. Section 3.1 presents an overview of the CPN model of the mobile phone UI software system. Section 3.2 describes how the individual features are modelled and how the similarities in communication patterns can be exploited through reuse of subpages of the CPN model.

3.1 Overview of the CPN Model

The CPN model has been hierarchically structured into 25 modules (subnets). The subnets are in CPN terminology also referred to as pages, and we will use this terminology throughout the paper.

The page Nokia, depicted in Fig. 1, is the top-most page of the CPN model and hence provides the most abstract view of the mobile phone UI software system. UIController, Servers, CommunicationKernel and Applications are all substitution transitions which means that the detailed behaviours are shown on the subpages with the corresponding names.
The four substitution transitions correspond to four concepts of the mobile phone UI software system: *applications*, *servers*, *UI controller*, and *communication kernel*.

Applications. The applications implement the features of the mobile phone, e.g., calls, games and calendar. The terms feature and application will be used interchangeably in the rest of the paper.

Servers. Servers manage the resources of the mobile phone, e.g., the battery, and implement the basic capabilities of the phone.

UI Controller. Applications make the feature available to the user via a user interface. The user interfaces are handled by the UI controller. Servers do

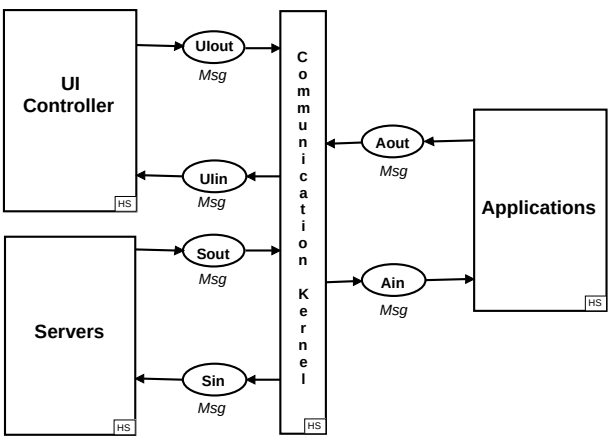


Fig. 1. Page Nokia

not have user interfaces. We will sketch the role of the UI controller by means of an example later in this section.

Communication Kernel. Servers and applications communicate by means of asynchronous message passing. The messages are sent through the communication kernel which implements the control mechanism and protocol used in the communication between the applications, servers and UI controller.

The focus in the MAFIA project is on features and feature interactions. Hence, we will not go into detail on the modelling of the UI controller, servers and communication kernel. However, we will informally sketch the role of the UI controller as it is necessary for the presentation of the CPN modelling of the applications (features).

The main role of the UI controller is to handle the user interfaces of the applications, i.e., to put the user interfaces of the applications on the display of the mobile phone and to forward key presses from the user to the right applications. Often several applications are active at the same time, for example, when the user is playing a game and a call comes in. We therefore need a priority scheme to determine which application will get access to the display and keys of the mobile phone. This is also the role of the UI controller. In Fig. 2 we give a small example to illustrate the operation of the UI controller. The example shows a scenario where the user of the mobile phone starts playing a game. While playing the game a call comes in. The scenario is shown using a Message Sequence Chart (MSC), which is automatically generated during simulation of the CPN model. The MSC contains a vertical line for the user, the UI controller and each of the applications and servers involved in the scenario (here the Game application, the Call application and the Call server). The marks on the vertical line corresponding to the UI controller indicate that the display is updated. A snapshot of the resulting display is shown below the MSC. The number next to the marks and displays indicates the correspondence. Arcs between the vertical lines correspond to messages sent in the mobile phone UI software system. The line numbers in the description below refer to the line numbers in the right-hand side of the MSC.

- The display of the mobile phone when the scenario starts is shown (line 1).
- The user presses the *left soft* key (the key just below the left-hand side of the display) to start a new game (line 2), and the Game application is notified (line 3).
- The Game application requests the display (line 4), the display is updated (line 5), and the Game application is acknowledged (line 6).
- An incoming call arrives (line 7), and the Call application requests the display (line 8).
- The Call application has higher priority than the Game application, hence the game is interrupted (line 9), the Game application acknowledges the interruption (line 10), and the display is granted to the Call application (line 11).
- Finally, the Call application is acknowledged (line 12).

The basic idea behind the model of the UI controller is that the UI controller maintains a structure of the displays (the *display structure*) that the applications

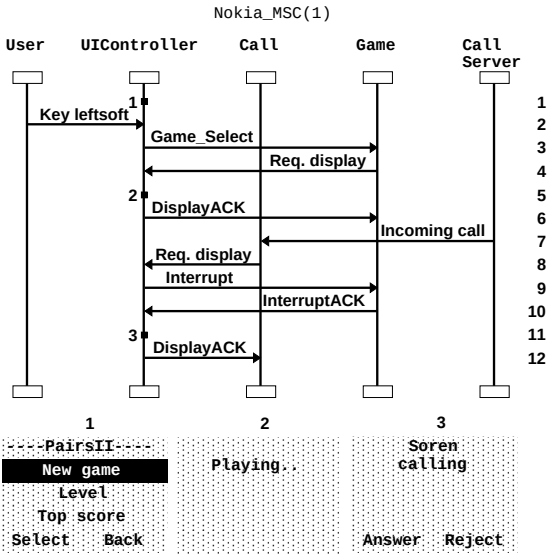


Fig. 2. Operation of the UI controller

in the system have requested to be put on the display of the mobile phone. Due to the limited UI resources of the mobile phone, not all of the display requests can be fulfilled. The displays are assigned a priority to assure that more important features, e.g., an incoming call, will interrupt less important features, e.g., a game. For a given priority the display structure contains a stack of display requests, i.e., if two applications request the display with the same priority the latest request will get the display. Hence, when applications request the UI controller to use the display of the mobile phone, the UI controller will put the display request in the display structure (on top of the stack in the corresponding priority). The UI controller will grant the display to the top of the first, i.e., highest priority, non-empty stack in the display structure. Figure 3 shows an example of the display structure where three priorities are shown: foreground, desktop and background with three, four and two display requests in the corresponding stacks, respectively. The display of the mobile phone will indicate that there is an incoming call (the top element of the stack with foreground priority).

3.2 Modelling of the Features

We will now go into more detail about the modelling of the features of the mobile phone. In the following we will use the Call feature as an example.

Figure 4 shows the detailed behaviour of the page Call modelling the Call feature. The Call feature is an essential feature of the mobile phone and is also the most complex feature included in the CPN model. To enhance readability the Call feature has been divided into a number of subpages. First we will give

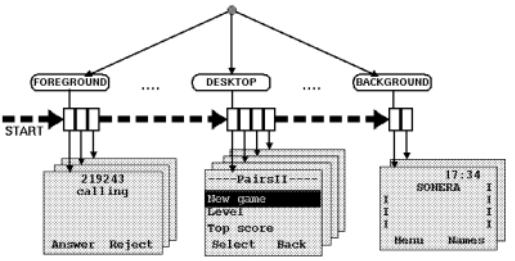


Fig. 3. Stack like structure of application displays

a brief overview of the Call feature, then we will concentrate on the part of the Call feature handling an incoming call.

The two places, Ain and Aout in the upper left corner in Fig. 4, model the message buffers to/from the Call application. It should be noted that there is an arc from Ain to each of the transitions in the page and an arc to Aout from each of the transitions in the page. However, to increase readability these arcs have been hidden in the figure.

The three places Idle, InCall (one active call) and In2Call (two active calls) model possible states of the Call application. Initially, the Call application is Idle. A new call can be established as either an Incoming Call or an Outgoing

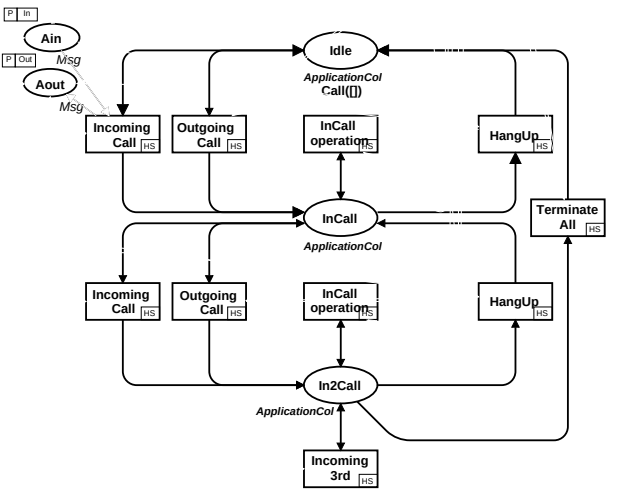


Fig. 4. Page Call

Call making the Call application change its state from Idle to InCall. When the Call feature is InCall, a second call can be established as either an Incoming Call or an Outgoing Call making the Call application change its state from InCall to In2Call. In InCall and In2Call a single call can be terminated (modelled by the substitution transitions Hangup) making the Call applications change its state to Idle or InCall, respectively. When the Call application is In2Call, both calls can be terminated (modelled by transition Terminate All) causing the Call application to change its state to Idle. When the Call feature has two active calls, i.e., is in state In2Call a third incoming call can be received (modelled by transition Incoming3rd). The third call cannot be established before one of the two active calls are terminated. Hence, the Call application will remain in In2Call.

Figure 5 shows the page IncomingCall which models the part of the Call application handling the reception of an incoming call. This is an example of the most detailed level of the CPN model of the features. As before, there is an arc from Ain to each of the transitions in the page and an arc to Aout from each of the transitions in the page. The arcs from/to the two places Ain and Aout have again been hidden to increase readability of the figure.

The places Idle, Indicating and InCall all have colour set ApplicationCol, which denotes an application and some internal data of the application (here the call and the internal data of the call, i.e., the number of the caller and the status of the call). These three places model the possible states of the Call feature related to the reception of an incoming call. We will explain the rest of the places (the three Init places) later.

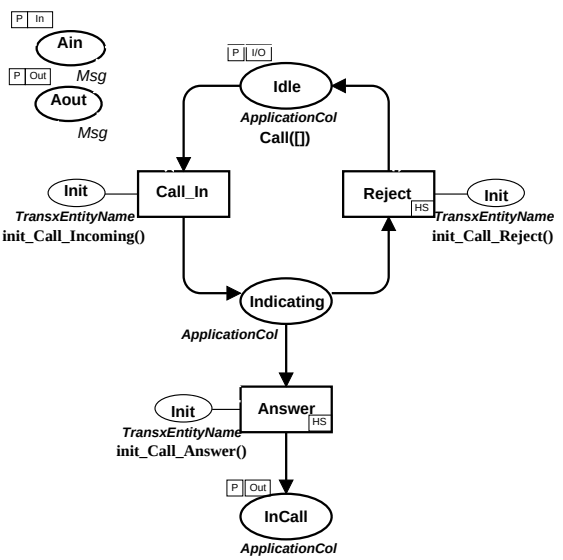


Fig. 5. Page IncomingCall

In Fig. 5 the arrival of an incoming call is modelled by the substitution transition Call_In, which causes the Call application to change its state from being Idle to Indicating. Now the call can be either answered or rejected by the user. The substitution transition Reject models the rejection of the incoming call and causes the Call application to change its state back to Idle. The substitution transition Answer models the answering of the call and causes the Call application to change its state from being Indicating to InCall. All the transitions in Fig. 5 are substitution transitions, hence the detailed behaviour is modelled on separate pages.

In the following we will give a detailed explanation of the substitution transition Call_In modelling an incoming call to the mobile phone. Figure 6 shows in detail what happens when an incoming call arrives in the mobile phone UI software system. The figure is a MSC generated from a simulation of the CPN model. Below we give an textual description. The line numbers refer to the line numbers in the right-hand side of the MSC.

The MSC contains a vertical line for the user, the UI controller and each of the applications and servers involved in the scenario (here the Call application, the Phonebook application, the Any key answer application, the Keyguard application and the Call server). When the simulation starts the phone is idle (line 1).

– The Call application is triggered by an incoming call (via the Call server) (line 2).

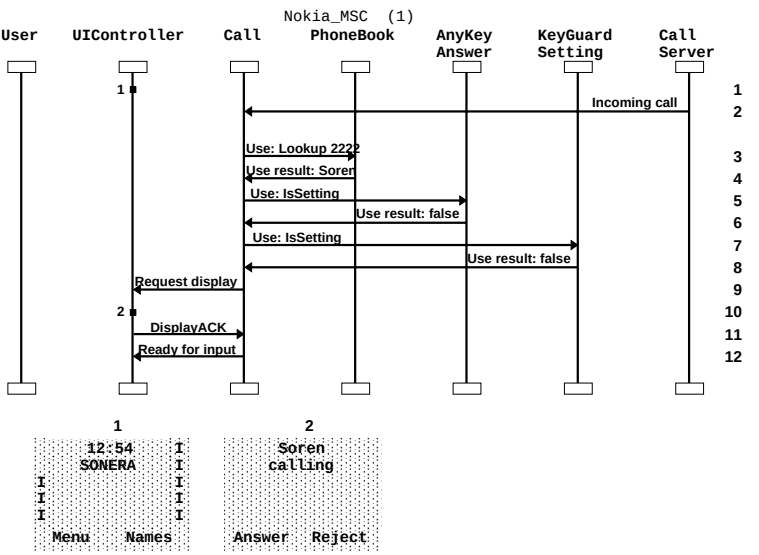


Fig. 6. An incoming call arrives in the mobile phone

– The Call application requests the UI controller, servers, and other applications in the mobile phone UI software system and uses the results to carry out its own task, i.e., to receive and notify the user about the incoming call.
  - The Phonebook application is requested to check whether the calling number is listed in the phonebook (in that case the name of the caller will be written in the display instead of the calling number) (lines 3-4).
  - The Any key answer application is requested to check whether the Any key answer setting is on (in that case the call application will allow the call to be answered using any key of the mobile phone, otherwise the call can only be answered using the *offhook* key, i.e., the key with the red phone) (lines 5-6).
  - The Keyguard application is requested to check whether it is active (in that case the call can only be answered using the *offhook* key independent of the state of the Any key answer application) (lines 7-8).
  - The UI controller is requested to put the user interface of the Call application on the display of the phone and to send information about user key presses to the Call application (line 9). The UI controller updates the display (line 10). Finally, the Call application is acknowledged (line 11). Note that the number was listed in the phonebook (the display indicates that Soren is calling), any key answer was not set, i.e., only information about the *offhook* key should be sent to the Call application (this cannot be seen from the figure).
– The call application changes its state and notifies the UI controller (line 12).

In the above description of the Call application's handling of the incoming call, the events and messages sent in the mobile phone UI software system have been divided into three parts: *trigger*, *request/result loop*, and *proceed*. In fact all state changes of the Call application (as well as the rest of the applications in the mobile phone UI software system) can be characterised by the same three steps. Hence, all of the state changes of the applications are modelled using instances of the same subpage modelling the general communication pattern sketched above. The page modelling the three steps is shown in Fig. 7.

The page GenerelSendReceive contains five *port places*: Start, End, Ain, Aout and Init. The two places Start and End (in the left and right of Fig. 7) are bound to the input place and the output place of the substitution transition (when the page is used for the Call_In substitution transition in Fig. 5, Start and End are bound to Idle and Indicating). Places Ain and Aout are bound to the message buffers to/from the application. Place Init is bound to the place connected to the substitution transition with a thin line (a line corresponds to an arc with arrowheads in both directions).

– The first step (trigger) is modelled by the Trigger transition modelling that the state change of the application is triggered by some event sent to the application via the Ain place causing the application to change its state from Start to Loop.

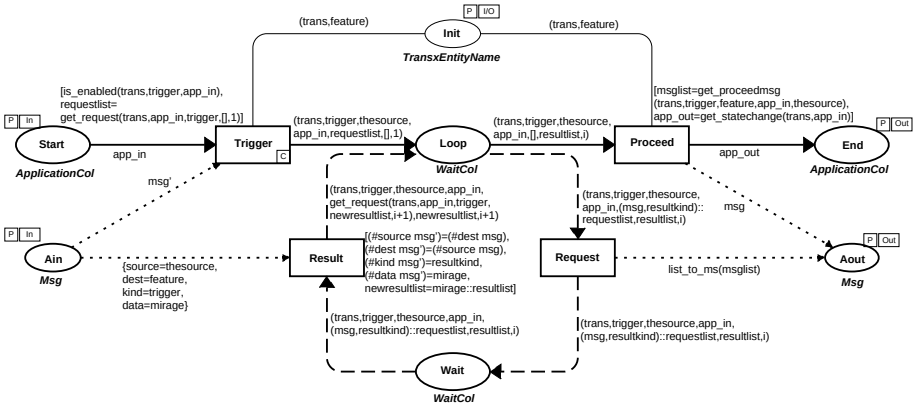


Fig. 7. Page GenerelSendReceive

- The second step (request/result loop) is modelled by the two transitions Request and Result (the loop in Fig. 7 with the dashed arcs). Request models the sending of a request to another entity in the system (an application, server or the UI controller) via the Aout buffer and causes the application to change its state from Loop to Wait. When a result appears on place Ain transition Result can occur, causing the application to change its state back to Loop. The loop can be executed several times until the application has no more requests.
- The third step (proceed) is modelled by the transition Proceed modelling that a message is sent to the Aout buffer. The application changes its state to End, thus modelling a complete state change of the application.

The concrete information about which messages trigger the event, which requests are sent, how the results are used by the application, and what change in data are performed, etc. is determined from the (initial) marking of the Init place. The same page (shown in Fig. 7) is used as a subpage for all of the substitution transitions modelling state changes of the applications with individual installations of the corresponding Init places. The reuse of the page shown in Fig. 7 means that even though the CPN model of the mobile phone UI software system only consists of 25 different pages it actually contains 98 page instances.

We have now presented the CPN model of the Call application to give an idea about the complexity of the CPN model and the efforts required to include a feature in the model. All features in the CPN model follow the same idea as the Call feature. Hence, when adding a new feature to the CPN model, a page modelling the overall state changes is provided together with a description of triggers, requests/results and proceed messages. This way of modelling the features reflects the way the features are currently documented in the written specifications of features in Nokia mobile phones. Here the features are described using a textual description of the possible states and state changes of the feature together with a description of the user interfaces and the use of other features and

the reactions of the feature to user key presses. Hence, the CPN model closely follows both the UI designers' and software developers' current understanding of features as well as the current available documentation of the features.

4 Simulations and Visualisation

We model the individual features of the mobile phone using the same ideas as described for the Call feature. The feature interactions are captured in the CPN model as the communication and interaction between the individual features in the CPN model. The UI designers and software developers who are developing new features will use the CPN model to identify and analyse the interaction patterns of their new features as well as existing features. One way of using the CPN model is by means of simulations; both interactively (step-by-step) and more automatically for investigations of larger scenarios. In this section we will present techniques which allow UI designers and software developers who are not familiar with Petri Nets to control and gain information from simulations without directly interacting with the underlying CP-net and its token game.

Two extensions have been made to the CPN model allowing the current state and behaviour of the CPN model to be visualised during simulations. Firstly, the state of the phone as the user observes it on the handset is visualised via an animation of the display. Secondly, the CPN model is extended with Message Sequence Charts (MSCs) which are automatically constructed as graphical feedback from simulations. MSCs were chosen to visualise the behaviour of the CPN model because diagrams similar to MSCs are already in use in the design process at Nokia. Therefore, MSCs allow the behaviour of the CPN model to be visualised in a way that is familiar to the UI designers and software developers. The use of domain-specific graphics and MSCs allow the state of the model to be visualised. However, the users of the CPN model are also interested in controlling the simulations without interacting directly with the underlying CP-nets, i.e., without explicitly selecting the transitions and bindings to occur. The CPN model has therefore also been extended with facilities for controlling the simulations without interacting with the underlying CP-nets.

In the following we will present techniques for visualising information about the simulation and for controlling the simulation using domain-specific graphics. Figure 8 shows how the domain-specific graphics appears as a single page in the CPN model of the mobile phone UI software system. In the following we will present the elements of the page shown in Fig. 8 in detail.

4.1 Animation of the Display

The CPN model is extended with a picture of a mobile phone. The picture is used both to present information about the simulation to the user and to get information from the user to control and guide simulations of the CPN model. The state of the phone as the user observes it via the handset is visualised via an animation of the display. The left-hand side of Fig. 8 shows a snapshot of the

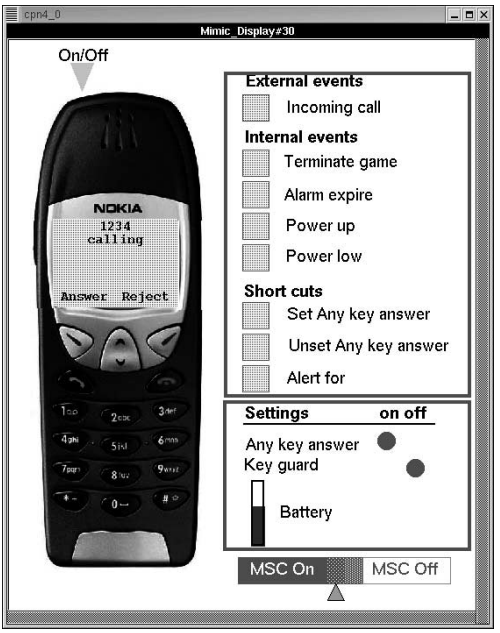


Fig. 8. The domain-specific graphics for the mobile phone UI software system

animation taken during a simulation of the CPN model. The snapshot shown corresponds to a state of the CPN model where the Call feature is Indicating. The user now has the possibility to either answer (transition Answer in Fig. 5) or reject (transition Reject in Fig. 5) the incoming call. The picture of the mobile phone is also used to control the simulation. By means of key presses (using mouse clicks) on the picture of the mobile phone, the user can control the simulation, i.e., start a game or answering an incoming call. In the following we will sketch how the use of domain-specific graphics has been integrated in the CPN model of the mobile phone UI software system.

The use of domain-specific graphics is implemented using the Mimic library [21] of Design/CPN, which allows graphical objects to be displayed, updated and selected during simulations. The animation in the left-hand side of Fig. 8 is constructed as number of layered objects:

- A background picture of a Nokia mobile phone is used to make the use of graphics look like a real Nokia mobile phone as it is known to the users of the CPN model.
- On top of the display of the background picture there are a number of regions (one for each area of the mobile phone display). Updating the contents of those regions allow us to animate the display of the mobile phone during simulation. We have implemented a function `update_display()` that scans through the display structure and finds the top element (display request) in the first non-empty stack, i.e., the one to be put on the display of the

mobile phone. The regions on top of the display of the background picture are updated with the contents of this "top most" display.
– On top of each of the keys of the background picture there is a region. We have implemented a function `get_input()` which enables the user to select one of the regions corresponding to the keys of the mobile phone and return information about which region, i.e., which key, is pressed.

The two functions `update_display()` and `get_input()` are called from code segments in the CPN model. Hence, the graphics is updated during simulations; user key presses are also read into the CPN model and can be used to control the simulations.

The animation of the display shows the state of the CPN model as the user of the handset observes it. Often the user is also interested in getting information about the internal state of the CPN model, e.g., the current personal settings of the mobile phone. Hence, the CPN model is also extended with graphics visualising the internal state of the phone. The lower part in the right-hand side of Fig. 8 shows how the internal state of the CPN model of the mobile phone UI software system is visualised using the Mimic library. The graphics visualise the state of Any key answer and Keyguard applications (whether they are *on* or *off*) and the amount of power in the battery. The graphics is updated dynamically during simulations to reflect the current state of the CPN model.

4.2 Controlling the Simulations

We have already described how the user can control the simulations by means of mouse clicks on the picture of the mobile phone. However, this often turns out to be insufficient. There are a number of events which are not generated as a result of user key presses, e.g., an incoming call to the mobile phone. The top-most part in the right-hand side of Fig. 8 shows how the CPN model has been extended with the possibility of generating such events by pressing buttons during simulations. The events are divided into three categories:

External Events i.e., events that are caused by the environment of the mobile phone (here incoming calls from the network).
Internal Events i.e., events that are generated by the mobile phone software without the influence of the user (here termination of a game, expiration of the alarm clock, battery level low warning, and battery charging).
Short Cuts allow the user of the CPN model to perform an operation without pressing a series of keys (here set or unset the Any key answer with a single key press instead of through the menu and change which phone numbers to alert for).

4.3 Message Sequence Charts

The animation of the display and the visualisation of the internal state of the mobile phone will provide information about the state of the CPN model as the

user of the mobile phone will observe it. However, the software developers are also often interested in gaining information about the UI software system at a more detailed level. This is obtained through the use of MSCs which capture the communication between applications, servers and the UI controller in the mobile phone UI software architecture. Figure 2 shows an example of a MSC automatically generated from a simulation of the CPN model. The MSC is described in detail in Sect. 3.

The MSC shown in Fig. 2 has a vertical line for both the user, the UI controller and each of the applications and servers involved in the scenario. During meetings with both UI designers and software developers we identified that the desired level of detail in the MSCs depends highly on the background of the user. Hence, we found it necessary to be able to customise the MSCs to the individual users of the CPN model. Hence, we now provide four possible levels of detail (see the lowest part in the right-hand side of Fig. 8). It should be noted that the MSCs can dynamically be turned on and off during simulations. Also the level of detail in the MSC can be changed dynamically. A small triangle below the buttons indicate which level is currently in use.

1. MSC On will generate detailed MSCs with vertical lines corresponding to the user, the UI controller as well as for all applications and servers in the system and arcs for all messages sent in the system.
2. The two buttons in the middle (between MSC On and MSC Off) generate MSCs in two intermediate levels of detail
   - The leftmost button generates MSCs like 1. but omits arcs corresponding to acknowledgement messages. The MSC in Fig. 2 is generated using this level of detail.
   - The rightmost button generates MSCs where only the communication between the user and the user interface is captured. This kind of MSC is mostly aimed at the UI designers who design the layout and the use of the display.
3. MSC Off turns off the use of MSCs.

In this section we have presented separate techniques for visualising information about the simulation and for controlling the simulation without interacting directly with the underlying CP-nets. It should be noted that during simulation of the CPN model, the page in Fig. 8 is the only page of the CPN model that needs to be visible to the user. Hence, the CPN model can be demonstrated and used without showing the underlying CPN model.

5 Related and Future Work

A number of published papers, e.g., [7,18,23], report on projects where CP-nets successfully have been used in industrial settings. Feature interactions have been studied in several application areas, e.g., telecommunications systems (see [15] for a survey), process planning [12], and computer-aided design [20]. To our knowledge there are no applications of CP-nets to feature interactions.

Especially in the area of telecommunication services much research work have been done on feature interactions and there is a biannual workshop on the topic [16,3]. Our work in the MAFIA project does not relate to what is traditionally known as *telecom feature interactions*; our problem is more general: how to coordinate concurrent interrelated services. However, we can draw some parallels. Much of the work in the area of feature interactions in telecommunications, e.g., [4,19,1], concentrate on the use of formal methods to automatically detect feature interactions. This is, however, not the aim of work the MAFIA project. In our work we focus on the development process of features, i.e, how the use of CP-nets to model the basic call model and the additional features can increase the level of understanding of the feature interactions and aid the future development of new features. Thus, we will categorise our approach to belong to the software engineering trend identified in [2] as one of the three major trends in the field of telecommunications services. However, the use of an underlying formal method, i.e., CP-nets, enables us to both obtain a formal description of the features and feature interactions and an environment for interactive exploration and simulation of the feature interactions. Furthermore, the construction of the CPN model as well as simulations were used to resolve inconsistencies in the specifications and to gain knowledge about features and feature interactions.

Based on the above, an interesting question is therefore whether the modelling approach of features developed in the MAFIA project can be used to automatically analyse and detect feature interactions. Using the Design/CPN Occurrence Graph Tool (OG Tool) [6] initial attempts have been done to evaluate the applicability of the CPN model for analysis purposes. State spaces have been generated for different combinations of features (including the features in isolation, i.e., the basic CPN model with only the analysed feature included). Not all combinations of features have been analysed and no attempts have been done to generate the state space for the full CPN model of the mobile phone UI software system presented in Sect. 3; only state spaces for selected combinations were generated with the goal of evaluating if (and how) state space analysis can be used to detect feature interactions in the CPN model developed in the MAFIA project.

In the following $\mathcal{S}_{f_1,f_2}$ denotes the full state space of the basic CPN model including the features f_1 and f_2. $\mathcal{S}_f \models P$ means that the property P can be verified from $\mathcal{S}_f$. One possible way of formally expressing that two features f_1 and f_2 interact is that for a property P we have that $\mathcal{S}_{f_1} \models P$ but $\mathcal{S}_{f_1,f_2} \not\models P$. We will use two properties P_1 and P_2 as examples:

P_1= There are no *deadlocks* and the initial state is a *home state*.

P_2= If a Call comes in and the Any Key Answer is set, then pressing any key on the mobile phone will answer the Call.

P_1 and P_2 can be formulated as *queries* in the OG tool. The answers to the queries can then be automatically determined by the OG tool when a state space has been generated. P_1 is general property of the CPN model and can be expressed as a standard query of a CPN model. P_2 relates to specific features

of the CPN model and can be formulated using temporal logic [9]. Properties expressed in temporal logic can be verified using the OG tool library ASK-CTL [8] which makes it possible to make queries formulated in a state and action oriented variant of CTL [5]. We will not go into detail with how the properties are expressed as queries. Instead we will show how P_1 and P_2 can be used to detect feature interactions in the CPN model of the mobile phone UI software system. Analysing different combinations of features we find that

1. $\mathcal{S}_{\mathsf{Call}} \models \neg P_1$ but $\mathcal{S}_{\mathsf{Call,AnyKeyAnswer,PhoneBook,KeyGuard}} \not\models \neg P_1$
2. $\mathcal{S}_{\mathsf{AnyKeyAnswer}} \models P_2$ but $\mathcal{S}_{\mathsf{AnyKeyAnswer,KeyGuard}} \not\models P_2$

The first feature interaction found is an interaction between the Call, Any key answer, Phonebook and Keyguard features. The interaction is a result of the Call application's use of other features in the mobile phone software system. Figure 6 shows how the Call applications requests the Phonebook application, the Any key answer application and the Keyguard application when an incoming call arrives at the mobile phone. Hence, a deadlock appears when the modelled Call application exists in isolation, i.e., without the Phonebook application, the Any key answer application and the Keyguard application.

The second feature interaction found is an interaction between the Any key answer and Keyguard features. The interaction appears as a result of the features' capability of modifying the behaviour of other features. Here the Keyguard application disables the Any key answer application to prevent the user from answering an incoming call by accident.

After having performed the basic analysis presented in this section we conclude that the CPN model developed in the MAFIA project seems applicable for analysis and detection of feature interactions; no major changes or adjustments are needed. However, we expect the state space of the full CPN model to be very large. Since the features of the mobile phone are asynchronous a possible direction for future analysis would be to use the stubborn set method [22] to reduce the size of the state space.

6 Conclusions

This paper describes the MAFIA project and one of its main activities: the construction of a CPN model of the mobile phone UI software system. Previous sections have reported on the CPN model and its construction. In this section we will conclude the paper by discussing how the CPN model was used within the MAFIA project and how the construction of the CPN model has influenced the development process of features in Nokia mobile phones.

The CPN model has been constructed in a period of six months and has been constructed in several iterative steps with more and more elaborated models each presented to the intended users of the CPN model, i.e., the UI designers and software developers. The fact that the CPN model was presented to users not familiar with Petri Nets meant that the CPN model was extended with

domain-specific graphics at a very early stage. The graphics was developed and extended in parallel with the underlying CP-net.

An important aspect of the CPN model developed in the MAFIA project is that the model is intended to be used in very different settings:

High Level (UI Oriented). UI designers design the features and their interactions at the level of the user interface of the mobile phone. The CPN model provides possibilities for simulations with feedback by means of the animation of the display and MSCs with detailed information about the flow of user interaction and snapshots of the contents of the mobile phone display. The UI designers can simulate the CPN model using only the page with Mimic graphics.

Conceptual Level. The architecture and protocol for the mobile phone UI software system is designed particularly to be well suited for development of features. The CPN model provides possibilities for simulations with more detailed feedback about the individual features and their interactions (messages sent between the individual applications, servers and the UI controller).

Low Level. Software developers specify and implement the new features of mobile phones. The CPN model is developed to make it possible to easily add new features in the model without changing the existing model. Hence, new features can be included in the CPN model and simulated with the features already included in the CPN model.

The construction of the CPN model of the mobile phone UI software system is only one of the activities in the MAFIA project. Other activities include development of a categorisation of feature interactions. During development of such a categorisation, the CPN model was used to experiment with the categories found. The CPN model have also been used to produce static figures (like the MSC in Fig. 2) to a document explaining feature interaction to UI designers and UI testers in Nokia. We have already described how the CPN model was constructed in close cooperation with UI designers and software developers of Nokia mobile phones. The UI designers and software developers also suggested features to be included in the CPN model to ensure that the categorisation of feature interactions developed in the MAFIA project is covered by features included in the CPN model. Simulations were used to understand the feature interactions and as a starting point for discussion.

The CPN model is intended to be used in future feature development in Nokia; both as a means for presenting and experimenting with the different features and feature interactions captured in the CPN model, but also as a framework for development of new features. New features can be designed and included in the CPN model and the behaviour and interactions with other features can be observed using simulations of the CPN model. However, even if the CPN model was not to be used in feature development, we find that the project group and the users involved in the development process have benefitted from the construction of the CPN model. During construction of the CPN model, a number of inconsistencies in the descriptions of the features have been identified. The modelling process has identified what kind of information about a feature's

interaction with other features need to be provided by the designer of the feature and what kind of problems need to be resolved when a feature is designed. In parallel with the MAFIA project the UI designers involved in the construction of the CPN model were designing a new feature for a Nokia product. During development of the feature, focus was pointed at interactions with other features based on experiences gained from the MAFIA project, and this resulted in a suggestion for new design templates and development practises.

The construction of the CPN model has also generated new ideas for the mobile phone UI architecture. The first version of the CPN model was constructed by the project group based on written documentation. A number of inconsistencies were found, and some of these could not be resolved before the first meeting with the users. Hence, a solution was chosen and presented to the users as a starting point for discussion. In fact, the chosen solution was not the correct solution with respect to the current implementation of the mobile phone UI software system. However, the solution modelled was found to be interesting and in fact a possible suggestion for a change in the mobile phone UI architecture to make it more oriented towards feature interactions.

In conclusion we can say that the MAFIA project has improved knowledge about features and feature interactions and has influenced an ongoing change of design practises of features in new products of Nokia mobile phones. The CPN model developed in the MAFIA project is intended to be used in future design of features of Nokia mobile phones. Furthermore, the modelling activities have raised interesting questions and ideas that can lead to design changes of the mobile phone UI architecture.

References

1. D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS. In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems*, volume VI, Amsterdam, May 2000. IOS Press. 309
2. M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. Submitted for publication. On-line version: `http://www.dcs.gla.ac.uk/~muffy/papers/calder-kolberg-magill-reiff.pdf`. 309
3. M. Calder and E. Magill. Feature Interactions in Telecommunications and Software Systems VI. IOS Press, 2000. 309
4. M. Calder and A. Miller. Using SPIN for Feature Interaction Analysis - a Case Study. In *Proceedings of SPIN 2001*, volume 2057 of *Lecture Notes in Computer Science*, pages 143–162. Springer-Verlag, 2001. 309
5. A. Cheng, S. Christensen, and K. Mortensen. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. In M. Spathopoulos, R. Smedinga, and P. Kozák, editors, *Proceedings of WODES'96*. Institution of Electrical Engineers, Computing and Control Division, Edinburgh, UK, 1996. 310
6. S. Christensen, K. Jensen, and L. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark. On-line version: `http://www.daimi.au.dk/designCPN/`. 309

7. S. Christensen and J. Jørgensen. Analysis of Bang and Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets. In P. Azéma and G. Balbo, editors, *Proceedings of ICATPN'97*, volume 1248 of *Lecture Notes in Computer Science*, pages 387–406. Springer-Verlag, 1997. 308
8. S. Christensen and K. H.Mortensen. *Design/CPN ASK-CTL Manual*. Department of Computer Science, University of Aarhus, Denmark, 1996. Online: `http://www.daimi.au.dk/designCPN/`. 310
9. E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986. 310
10. Design/CPN Online. `http://www.daimi.au.dk/designCPN/`. 295
11. H. Genrich. Predicate/Transition Nets. In K. Jensen and G. Rozenberg, editors, *High-level Petri Nets*, pages 3–43. Springer-Verlag, 1991. 295
12. J.-S. Hwang and W. A. Miller. Hybrid Blackboard Model for Feature Interactions in Process Planning. *Computers and Industrial Engineering*, 29(1–4):613–617, 1995. 308
13. ITU (CCITT). Recommendation Z.120: MSC. Technical report, International Telecommunication Union, 1992. 296
14. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992. 295
15. D. O. Keck and P. J. Kuehn. The Feature and Service Interaction Problem in Telecommunication Systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998. 308
16. K. Kimbler and L. G. Bouma. Feature Interactions in Telecommunications and Software Systems V. IOS Press, 1998. 309
17. L. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, December 1998. 295
18. L. Lorentsen and L. Kristensen. Modelling and Analysis of a Danfoss Flowmeter System. In M.Nielsen and D.Simpson, editors, *Proceedings of ICATPN'2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 346–366. Springer-Verlag, 2000. 308
19. M. Nakamura, Y. Kakuda, and T. Kikuno. Feature Interaction Detection using Permutation Symmety. In K. Kimbler and L. G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems*, volume V, pages 187–201, Amsterdam, September 1998. IOS Press. 309
20. D.-B. Perng and C.-F. Chang. Resolving Feature Interactions in 3rd Part Editing. *Computer-Aided Design*, 29(10):687–699, 1997. 308
21. J. L. Rasmussen and M. Singh. *Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual*. On-line version: `http://www.daimi.au.dk/designCPN/`. 306
22. A. Valmari. Error Detection by Reduced Reachability Graph Generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988. 310
23. J. Xu and J. Kuusela. Analyzing the Execution Architecture of Mobile Phone Software with Colored Petri Nets. *Software Tools for Technology Transfer*, 2(2):133–143, December 1998. 295, 296, 308

Analysing Infinite-State Systems by Combining Equivalence Reduction and the Sweep-Line Method

Thomas Mailund

Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK,
mailund@daimi.au.dk

Abstract. The sweep-line method is a state space exploration method for on-the-fly verification aimed at systems exhibiting progress. Presence of progress in the system makes it possible to delete certain states during state space generation, which reduces the memory used for storing the states. Unfortunately, the same progress that is used to improve memory performance in state space exploration often leads to an infinite state space: The progress in the system is carried over to the states resulting in infinitely many states only distinguished through the progress. A finite state space can be obtained using equivalence reduction, abstracting away the progress, but in its simplest form this removes the progress property required by the sweep-line method. In this paper we examine a new method for using equivalence relations to obtain a finite set of classes, without compromising the progress property essential for the sweep-line method. We evaluate the new method on two case studies, showing significant improvements in performance, and we briefly discuss the new method in the context of *Timed Coloured Petri Nets*, where the "increasing global time" semantics can be exploited for more efficient analysis than what is achieved using a "delay" semantics.

Keywords: Infinite-State Systems, State Space Analysis, Reduction Techniques, Timed Coloured Petri Nets.

1 Introduction

State space analysis is one of the main tools for analysis of Petri nets. The basic idea behind state space analysis is to construct a directed graph, called the *state space*, in which the nodes correspond to the set of reachable markings, and the arcs correspond to transition occurrences. The state space graph represents all possible interleaved executions of the model in question, and through analysis of this graph, most interesting properties of the model can be investigated. Unfortunately even relatively simple systems can have an astronomical number, or even an infinite number, of reachable states, preventing the construction of the full state space graph. To lessen this, so called, *state explosion problem*, a number of reduction methods have been developed. Reduction methods typically exploit

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 314–334, 2002.
© Springer-Verlag Berlin Heidelberg 2002

certain characteristics of the system being analysed either to represent the full state space in a condensed form, or to represent only a subset of the full state space. The reduction is done in such a way that the analysis questions of interest can still be determined from the reduced state space.

The *sweep-line method* [7] is such a reduction method. It is aimed at systems for which it is possible to define a *progress measure*, based on the states of the system. The progress measure gives us a conservative approximation of the reachablitity relation that makes it possible to delete certain states during state space exploration and to reclaim memory. When calculating the state space, we process the states in turn to calculate their successors. We examine the successors to determine which of them represent a state that we have already processed, and which represent a new state. If the state has already been processed, we stop processing from that point, if not, we store it for later processing. The nature of progress, as formally defined later in this paper, ensures that from a set of states, it is never possible to reach a state with a smaller progress value than the minimal value among the states in the set. In state space exploration this property can be used to safely *garbage collect* states with progress value less than the minimal among the unprocessed states. None of these states are reachable from the unprocessed states, so none of them are needed for comparison with newly processed states. Intuitively, the idea is to drag a *sweep-line* through the full state space, calculate the reachable states in front of the sweep-line and delete states behind the sweep-line. For finite state systems, the sweep-line method terminates after having processed each state exactly once, making the method suitable for checking safety properties. The progress measure will often be specific for the model under consideration, e.g. it is obtained from structural properties such as siphons and traps. However, for some modelling languages a progress measure is embedded in the formalism. An example is *Timed Coloured Petri Nets (TCPN)* [12,15] where the global clock can be used as a progress measure for states.

The sweep-line method was originally developed for analysis of Timed Coloured Petri Nets, where the progress measure is obtainable from the *global clock*. However, as observed in [12] and [8], the time concept of the formalism often leads to infinite state spaces. The progress is carried over into the state space, distinguishing otherwise identical states. As a consequence, most state space analysis of Timed Coloured Petri Nets has had to rely on *partial state spaces*, i.e., finite subsets of the full state space, and the sweep-line method has been used to increase the number of states that can be analysed [7,4]. Equivalence reduction [12] can be used to factor the infinite number of states, resulting from the progress, into a finite graph. This was done for Timed Coloured Petri Nets in [8] and Interval Timed Coloured Petri Nets [17] in [1]. However, combining equivalence reduction and the sweep-line method is not straightforward. The progress measure is essential for the sweep-line method while removing the progress, using the equivalence reduction, is the approach taken to obtain a finite state space. The new method we suggest in this paper is using the sweep-line method to explore the state space of such infinite state systems, until all reachable equiv-

alence classes (corresponding to all reachable nodes in the equivalence reduced state space) have been examined at least once.

The key idea underlying the new method is to use the sweep-line method to explore the state space, but at the same time keep track of the equivalence classes already examined. When we later find a state we recognise as being in the same equivalence class as a previously seen state, we stop processing from that point. Since we have already explored the states reachable from an equivalent state, processing states reachable from the new state will only lead to states equivalent to states reachable from the first state. The new method differs from the equivalence reduction method in that we do not keep *all* equivalence classes that we see during the state space exploration stored in memory, instead, we only keep a subset of the seen equivalence classes, which combined with the garbage collection in the sweep-line method reduces the memory usage compared with the original equivalence reduction. The new method also differs from the straightforward combination of the sweep-line method and equivalence reduction, briefly described in [4], where we require that the equivalence relation is compatible with the progress measure, i.e. equivalent states must have the same progress value. When this is the case, the progress measure is also an approximation of the reachability relation on the equivalence reduced state space, and the sweep-line method can immediately be applied. In the setting we consider in this paper, however, the equivalence relation abstracts away from the progress in the states, and the relation between progress values and reachability in the equivalence reduced state space. Though developed mainly for analysis of TCPN, we describe the method in more general terms, but return to TCPN at the end of Sect. 6.

The paper is organised as follows: Section 2 introduces a small example that will be used for describing the techniques of the new method. Section 3 briefly describes the original sweep-line method, and Sect. 4 describes equivalence reduction. Section 5 then discuss the new combination of the sweep-line method and equivalence reduction. Section 6 describes experimental results obtained with a prototype implementation of the new method, and finally Sect. 7 concludes the paper.

2 An Example

In this section we introduce a small example, which we will use to explain the new technique. The example is a Hierarchical Coloured Petri Net [11] model of a stop-and-wait protocol from the data-link control layer of the OSI architecture [2], and it is a slightly modified version of the model described in detail in [15].[1] The protocol consists of two main modules, a sender and a receiver module that communicate over an underlying communication channel, see Fig. 1.

The sender module, seen in Fig. 2, accepts data packets for transmission from a higher protocol layer. It then assigns a sequence number to the packet and transmits the data frame using the underlying protocol layer. The receiver

[1] The modifications consist of adding a parameterised buffer-capacity to the underlying network.

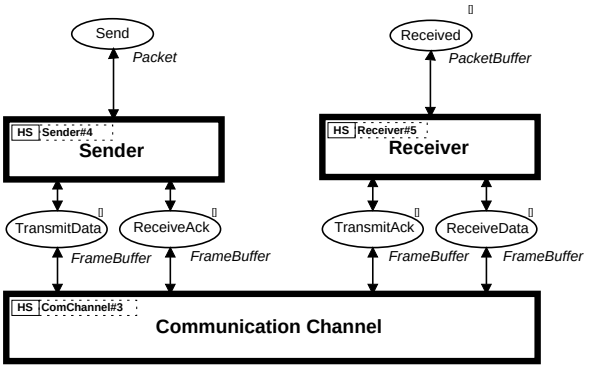


Fig. 1. Top-level view of the stop-and-wait protocol. The figure shows the main modules of the model, consisting of the Sender module, the Receiver module, and the underlying Communication Channel

module, seen in Fig. 3, receives frames from the underlying layer and, depending on the sequence number, either passes the data to the higher layer, or discards the frame. In any case, the receiver returns an acknowledgement frame containing the next expected sequence number to the sender.

The purpose of introducing this example is to illustrate the techniques used in the new method proposed in this paper. The model should not be thought of as a typical representative of the class of systems the new method is aimed at. The infinite number of states in this system is, in a sense, artificial. The infinite is a consequence of unbounded sequence numbers, something that could never be found in a real implementation of a stop-and-wait protocol. Even ignoring this, one would not need to use the proposed method to analyse a model of such a relatively small size. An equivalence reduced state space of a manageable size can still be obtained for most configurations. However, the model exhibits the characteristics exploited by the new method, it is a model of a fairly well-known system, and it will serve nicely as an introduction to the new technique.

3 The Sweep-Line Method

In the stop-and-wait protocol we can observe that the sequence number assigned to new data packets, and expected at the receiver, is never decreased. Rather, with each successfully received packet, the receiver increments the expected sequence number by one. Thus, these sequence numbers can be thought of as a measure of how much the system has progressed. Since the progress measure is never decreased we know that from a state s we can never reach another state s' where the progress (receiver sequence number) of s' is less than the progress (receiver sequence number) of s. Exploiting this property is the basic idea underlying the sweep-line method [7].

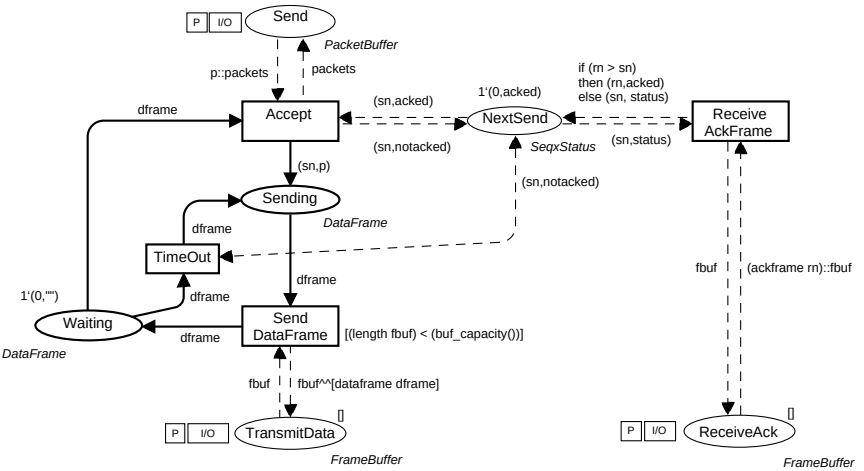


Fig. 2. The sender module of the stop-and-wait protocol. Data packets to be transmitted are taken from the place Send and assigned a sequence number from the place NextSend. Data frames are then transmitted via the underlying communication channel, when the buffer capacity of the underlying layer permits this. If no acknowledgement is received after a certain delay the sender times out and retransmits the frame. When an acknowledgement with the appropriate sequence number is received, the sender sequence number is incremented, and the sender can accept the next packet

In this and the following sections we will use this notation: The *state space* of a Petri net is a tuple (S, T, Δ, s_0), where S is the set of *states* (or markings), T is the set of *transitions* (or for Coloured Petri Nets the set of *binding elements*), $\Delta \subseteq S \times T \times S$ is the *transition relation*, satisfying $(s, t, s') \in \Delta$ (written $s \xrightarrow{t} s'$) iff t is enabled in s and the occurrence of t in s leads to s', and s_0 is the *initial state* (the initial marking) of the net. A state s_n is *reachable* from a state s_1, written $s_1 \rightarrow^* s_n$, iff there exists an *occurrence sequence* leading from s_1 to s_n, i.e. states $s_2, s_3, \ldots, s_{n-1}$ and transitions $t_1, t_2, \ldots t_{n-1}$ such that $(s_i, t_i, s_{i+1}) \in \Delta$ for $1 \leq i \leq n-1$. For a state s, $reach(s) = \{\, s' \in S \mid s \rightarrow^* s' \,\}$ denotes the set of states reachable from s. The set of *reachable states* is $reach(s_0)$.

A *progress measure* specifies a *partial order* $(O, \sqsubseteq)$ on the states of the model. A partial order $(O, \sqsubseteq)$ consists of a set O and a relation $\sqsubseteq$ over O which is reflexive, transitive, and antisymmetric. Moreover, the partial order is required to preserve the reachability relation $\rightarrow^*$ of the model:

Definition 1 (Def. 1 [7]). *A* ***progress measure*** *is a tuple* $(O, \sqsubseteq, \psi)$ *such that* $(O, \sqsubseteq)$ *is a partial order and* $\psi : S \rightarrow O$ *is a mapping from states into* O *satisfying:* $\forall s, s' \in S : s \rightarrow^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$.

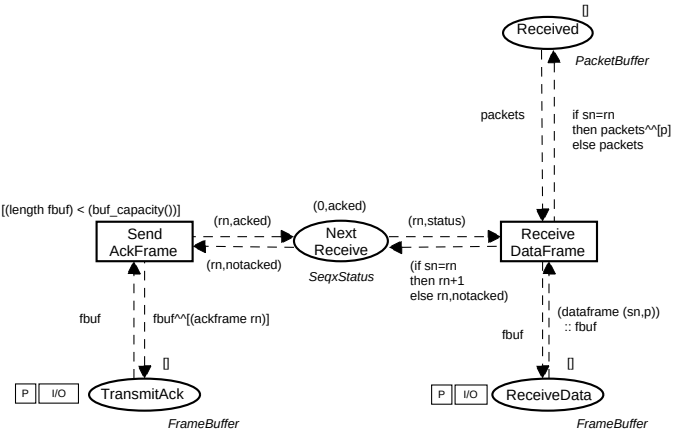


Fig. 3. The receiver module of the stop-and-wait protocol. Data frames are received from the underlying communication channel, and if the sequence number matches the one expected, the data frame is delivered to the upper protocol layer. Whether the sequence number matches or not, an acknowledgement containing the next expected sequence number is returned over the underlying channel, when the buffer capacity of the underlying layer permits this

The key property of a progress measure is the fact that $\psi(s) \not\sqsubseteq \psi(s')$ implies $s' \notin \mathit{reach}(s)$. It is this property that permits garbage collection of processed states during state space exploration. In conventional state space generation, all states are kept in memory in order to recognise previously processed states. However, states which have a progress measure which is strictly less than the minimal progress measure of those states waiting to be processed, can never be reached again. It is therefore safe to delete such states. Saving memory by deleting such states is the basic idea underlying the sweep-line method. The role of the progress measure is to be able to recognise such states. The pseudo-code shown in Fig. 4 describes the algorithm for the sweep-line method. Essentially, the algorithm is the usual state space generation algorithm (see e.g. [12]) extended with garbage collection and a processing order determined by the progress measure.[2]

Figure 5 (taken from [7]) illustrates the intuition behind the sweep-line method. The two grey areas show the states kept in memory. Some of these have been processed (light grey), i.e., their successor states have been calculated, and some are still waiting to be processed (dark grey). There is a *sweep-line* through the stored states separating the states with a progress measure which is strictly less than the minimal progress measure among the unprocessed states, from the states which have progressed further than the minimal unprocessed states. States

[2] The edges in the state space are ignored in the algorithm to keep the description minimal, but nothing in the sweep-line method prevents examining edges as well as nodes.

```
 1: UNPROCESSED ← {s_0}
 2: NODES.ADD(s_0)
 3: while ¬ UNPROCESSED.EMPTY() do
 4:   s ← UNPROCESSED.GETMINELEMENT()
 5:   for all (t, s') such that s -t-> s' do
 6:     if ¬(NODES.CONTAINS(s')) then
 7:       NODES.ADD(s')
 8:       UNPROCESSED.ADD(s')
 9:     end if
10:   end for
11:   NODES.GARBAGECOLLECT(min{ψ(s) | s ∈ UNPROCESSED})
12: end while
```

Fig. 4. The sweep-line method. The structure UNPROCESSED keeps track of the states for which successors are still to be calculated. In each iteration (lines 3-12) a new unprocessed state is selected (line 4), such that this state has a minimal progress value among the states in UNPROCESSED. The nodes explored so far are stored in NODES, and nodes are only added to UNPROCESSED (lines 6-9) if they are not already in NODES. After a node has been processed, nodes with a progress value strictly less that the minimal progress value among the states in UNPROCESSED can be deleted (line 11). The states are examined in an order determined by the progress measure in order to be able to garbage collect a node as soon as possible

strictly to the left of the sweep-line can never be reached from the unprocessed states and can therefore safely be deleted. As the state space generation proceeds, the sweep-line will move from left to right, while we calculate new states in front of the sweep-line and delete old states behind the sweep-line. All states on the sweep-line have the same progress value. By processing states ordered by their progress values, the number of states kept in memory (the two grey areas in Fig. 5) can be kept minimal.

The sweep-line method deletes nodes shortly after having created them. Hence, to obtain verification results, properties must be checked on-the-fly. Since the sweep-line method visits all reachable states it is ideal for checking safety properties.

4 Equivalence Reduction

The *equivalence reduction method* [12] is a generalisation of the symmetry method [9,10,13]. The basic idea behind equivalence reduction is to group equivalent states into equivalence classes and construct a state space of the equivalence classes rather than the original states. This reduced state space can then be used to verify properties compatible with the equivalence relation.

In the stop-and-wait protocol, if we abstract away from the actual data in the individual data frames, the only thing distinguishing the data frames are

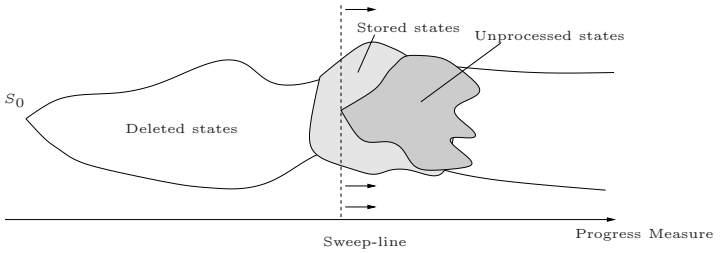


Fig. 5. The sweep-line method. We drag the sweep-line through the state space, calculating the reachable states in front of the sweep-line and deleting states behind the sweep-line

the sequence numbers. Two states of the protocol can be considered equivalent if the sequence numbers of the different tokens match, regardless of the values of the associated data, since the behaviour of the protocol only depends on the sequence numbers and not the data being transmitted. Furthermore, since the protocol only needs to recognise whether a received frame contains the correct sequence number, but otherwise does not depend on the actual value of the sequence number, we can consider two states equivalent if they agree on the relative distance between the sequence numbers, though not necessarily on the absolute values. So, for example, the state:[3]

```
NextSend:      1‘(1,notacked)     NextReceive:  1‘(1,acked)
Waiting:       1‘(1,"data1")      ReceiveData:  1‘[]
ReceiveAck:    1‘[]               TransmitAck:  1‘[]
TransmitData:  1‘[(1,"data1")]
```

can be considered equivalent to the state

```
NextSend:      1‘(3,notacked)     NextReceive:  1‘(3,acked)
Waiting:       1‘(3,"data3")      ReceiveData:  1‘[]
ReceiveAck:    1‘[]               TransmitAck:  1‘[]
TransmitData:  1‘[(3,"data3")]
```

since the second state can be obtained from the first by increasing all sequence numbers by two, i.e. the relative distance between sequence numbers in the two states are equal, and since we ignore the data being transmitted, i.e. `data1` is considered indistinguishable from `data3`. Unless we restrict the number of packets to be sent by the protocol, the state space of the model will be infinite, due to the increasing sequence numbers. However, if we only consider the relative distance in sequence numbers as being relevant we can obtain a finite representation of the state space, which contains all reachable equivalence classes of the infinite full state space.

[3] In this example, only the marking of places in the sender and receiver modules are shown

We will use the following notation: Let $\approx$ be an equivalence relation over the set of states S of some model. We let $S_\approx$ denote the set of all equivalence classes for $\approx$. For a state $s \in S$, $[s]_\approx \in S_\approx$ denotes the equivalence class of $\approx$ containing s. We will write $[s]$ instead of $[s]_\approx$ when $\approx$ is known from context. In the more general equivalence reduction method (as the one found in [12]) the reduction is specified by both an equivalence relation on the set of model states and an equivalence relation on the set of transitions, but for our purpose, the equivalence relation on transitions is not strictly needed. There is nothing in the new method that prevents it though, and the only reason it has been left out is to simplify the presentation.

Not all equivalence relations are useful for equivalence reduction. To ensure that the equivalence graph can be used to reason about the model, we require that the equivalence is *consistent* with the model:

Definition 2 (Def. 2.2 [12]). *An equivalence relation $\approx \subseteq S \times S$ is **consistent** iff the following condition holds for all states $s_1 \in S$, $s_2 \in [s_1]_\approx$, and transitions $t \in T$:*

$$s_1 \xrightarrow{t} s_1' \;\Rightarrow\; \exists s_2' \in [s_1']_\approx : s_2 \xrightarrow{t} s_2'.$$

The equivalence reduced state space of a model is then constructed from the equivalence classes of a consistent equivalence relation:

Definition 3 (Def. 2.3 [12]). *Let $\approx$ be a consistent equivalence relation. The **equivalence reduced state space** of a model with state space (S, T, Δ, s_0) is the tuple $(S_\approx, T, \Delta_\approx, [s_0]_\approx)$, where $([s]_\approx, t, [s']_\approx) \in \Delta_\approx$ iff $(s, t, s') \in \Delta$.*

The equivalence reduced state space has a node for each reachable equivalence class, and it has an edge between two nodes iff there is a state in the equivalence class of the source node in which a transition is enabled, and whose occurrence leads to a state in the equivalence class of the destination node. The algorithm for constructing the equivalence reduced state space is shown in Fig. 6.

The equivalence reduced state space is usually implemented by representing each equivalence class by a representative state for the classes. Constructing the state space then follows the same procedure as constructing the full state space, with one exception. Whenever a new state is to be inserted into the graph it is checked whether an *equivalent* state is already included in the graph. The construction can often be optimised by using a canonical representative, i.e., a unique representative for each equivalence class. The check then amounts to transforming the new state into this unique representative for the equivalence class and then check (using ordinary equality) whether the resulting state has already been included in the state space. In the stop-and-wait protocol we can obtain a canonical representation for the equivalence class of a state, by subtracting the value of the next expected sequence number from all sequence numbers in the state, and setting all data packet values to a dummy value. This will set the next expected sequence number to zero while preserving the relative distances of sequence numbers, and will make the data frame values indistinguishable.

```
 1: UNPROCESSED ← {s_0}
 2: NODES.ADD([s_0])
 3: while ¬ UNPROCESSED.EMPTY() do
 4:   s ← UNPROCESSED.GET()
 5:   for all (t, s') such that s -t-> s' do
 6:     if ¬(NODES.CONTAINS([s'])) then
 7:       NODES.ADD([s'])
 8:       UNPROCESSED.ADD(s')
 9:     end if
10:   end for
11: end while
```

Fig. 6. Exploration of the equivalence reduced state space. The algorithm resembles the usual state space exploration algorithm, but instead of storing the previously explored *nodes*, the corresponding *equivalence classes* are stored, and a state is only considered unprocessed if its equivalence class is not stored among the nodes

Since, at least intuitively, the equivalence reduced state space contains equivalence classes rather than the actual states, the reduced state space can only be used to verify properties which does not distinguish between states in the same equivalence class. We therefore require that all *state properties*, predicates over states $\mathsf{sp} : S \to \{\mathtt{tt}, \mathtt{ff}\}$, under consideration are *compatible* with the equivalence relation.

Definition 4. *For an equivalence relation* $\approx \subseteq S \times S$ *and state property* sp *we say that* sp *is* ***compatible*** *with the relation if* $s \approx s' \Rightarrow \mathsf{sp}(s) = \mathsf{sp}(s')$.

Using equivalence reduction it is possible to check properties constructed from state properties compatible with the equivalence relation used for the reduction. In the case of the stop-and-wait protocol, with the equivalence relation described earlier in this section, the equivalence reduced state space can be used to check properties not referring to the data being transmitted across the network and not referring to absolute sequence numbers, but potentially referring to relative differences between sequence numbers.

For the rest of this paper we will only consider models with finite equivalence reduced state spaces, though not necessarily finite (full) state spaces. The method described in the following section is aimed at exploiting progress to analyse models with an infinite state space, but finite equivalence reduced state space.

5 Using Equivalence Classes in the Sweep-Line Method

The new method extends the sweep-line method with elements from equivalence reduction. Similar to the equivalence reduction method, we store representatives for equivalence classes that we have seen during state space exploration, and

when we later find a state contained in a previously seen equivalence class, we stop processing from that point. Since we have already explored the states reachable from an equivalent state, processing states reachable from the new state will add no further information. However, in the new method we do not store all equivalence classes seen during processing. Instead, the new method only stores a subset of the equivalence classes seen during sweep-line exploration, reducing the peak memory usage. For selecting the equivalence classes to keep throughout the state space sweep, we use a predicate $p : S \to \{\mathtt{tt}, \mathtt{ff}\}$. For states s with $p(s) = \mathtt{tt}$, we keep $[s]$. For states s with $p(s) = \mathtt{ff}$, we do not keep $[s]$.

Selecting the right predicate for determining which equivalence classes to store is crucial for the method. If too many classes are stored the memory usage reduction is limited (if there is a reduction at all), and in the worst case there may not be enough computer resources to explore the state space. On the other hand, if too few classes are stored, there might exist occurrence sequences in which no previously processed equivalence class is recognised, in which case the exploration will not terminate. Under the assumption that there are only finitely many equivalence classes of states, any infinite occurrence sequence will at some point reach an equivalence class that was previously encountered. But since we do not store all equivalence classes, we do not necessarily recognise this state. To ensure termination of the method, a predicate must be chosen such that along all infinite occurrence sequences, at some point a state will be recognised as being equivalent to a previously seen state.

In models with progress, occurrence sequences come in two forms: *progress-bounded* and *progress-unbounded*.

Definition 5. *An infinite occurrence sequence* $s_1 \to s_2 \to s_3 \cdots$ *is said to be* ***progress-unbounded*** *if* $\forall i \in \mathbb{N}\ \exists j > i \ : \ \psi(s_i) \sqsubset \psi(s_j)$. *Any occurrence sequence that is not progress-unbounded is said to be* ***progress-bounded****.*

In the stop-and-wait protocol, a progress-unbounded occurrence sequence is an infinite sequence of packets transmitted and, after a finite number of retransmissions, successfully received. A progress-bounded sequence, on the other hand, would be either a finite sequence or an infinite sequence of transmissions where after a finite number of successful transmissions either all data transmissions or all acknowledgements are lost on the communication channel.

All progress-bounded occurrence sequences are either finite or will, after a finite number of steps, reach a loop in which all states have the same progress measure. Finite sequences will obviously not cause the method not to terminate. For infinite progress-bounded occurrence sequences we can observe, that since states with a given progress measure cannot be deleted before the sweep-line has moved beyond that progress measure, all states with the same progress measure must be stored in the state space at one point. Cycles of states with the same progress measure will therefore always be detectable in the state space, and thus for progress-bounded infinite occurrence sequences we will be able to recognise repeated states.

For progress-unbounded occurrence sequences we require that an infinite number of states in the sequence satisfy the predicate. Assuming that the equiv-

```
 1: Unprocessed ← {s_0}
 2: Nodes.Add(s_0)
 3: if p(s_0) then
 4:   EquivClasses.add([s_0])
 5: end if
 6: while ¬ Unprocessed.Empty() do
 7:   s ← Unprocessed.GetMinElement()
 8:   for all (t, s') such that s -t-> s' do
 9:     if ¬(Nodes.Contains(s')) ∧ ¬(EquivClasses.Contains([s'])) then
10:       Nodes.Add(s')
11:       Unprocessed.Add(s')
12:       if p(s') then
13:         EquivClasses.Add([s'])
14:       end if
15:     end if
16:   end for
17:   Nodes.GarbageCollect(min{ψ(s) | s ∈ Unprocessed})
18: end while
```

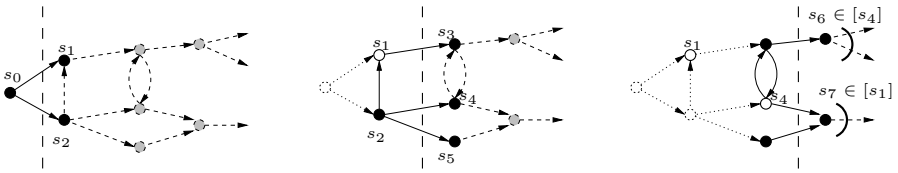
Fig. 7. The processing of new states in the new method. The sweep-line method is extended with a set EquivClasses of equivalence classes, in which all equivalence classes of states satisfying the predicate p is inserted. A new node is only processed if it is not already processed, nor has an equivalent class which is already processed

alence reduced state space only contains a finite number of equivalence classes this will ensure that after a finite number of steps a state equivalent to a previously encountered state satisfying the predicate will be reached. Since we store states satisfying the predicate, we will be able to recognise this situation and stop processing at that state. We define a *cut-off predicate* to be a predicate satisfying this for all progress-unbounded occurrence sequences.

Definition 6. *A predicate $p : S \to \{\mathtt{tt}, \mathtt{ff}\}$ is said to be a* ***cut-off predicate*** *if all progress-unbounded occurrence sequences starting from the initial marking contain an infinite number of states satisfying p.*

Given a cut-off predicate we will be able to sweep through the state space and terminate after all equivalence classes have been visited at least once. The method works as follows. The state space is explored using the sweep-line method, except that for each new state processed we check whether the state itself or a state equivalent to the new state is stored, in which case we stop processing along this path, if not we check whether the cut-off predicate is satisfied, in which case we save the equivalence class. The algorithm in Fig. 7 shows the processing of new states. Figure 8 illustrates the method on a small example.

A major drawback of the method is the fact that automatically deriving a cut-off predicate from the model is not immediately possible. Even checking



(a) The algorithm after the initial state has been examined and the two successors found and stored as unprocessed

(b) The states s_1 and s_2 have been examined and their successors stored as unprocessed. The state s_1 satisfied the predicated and its class was stored

(c) More states have been examined. State s_4 also satisfied the predicate, and its class is stored. The two successor states are both found to be in previously processed classes

Fig. 8. Illustration of the new method. The sweep-line (dashed vertical line) is dragged through the state space as in the sweep-line method, but as states are processed, the cut-off predicate is evaluated, and if a state satisfies the predicate its equivalence class is kept in memory. In the state space above, the states s_1 and s_4 satisfy the predicate and consequently once they have been processed we will be able to recognise when new states are equivalent to either of those. In (c) s_6 is found to be in $[s_4]$ and s_7 in $[s_1]$ and therefore neither s_6 nor s_7 is added to UNPROCESSED, and therefore further exploration is cut off at these points

that a given predicate is a cut-off predicate will in, the general case, require analysis similar to the analysis that we need the predicate for in the first place. Fortunately, a modeller-supplied predicate can at most cause one-sided errors, in the sense that a poorly chosen predicate can cause the method not to terminate but not cause the method to avoid reachable equivalence classes. Thus, even with a poorly chosen predicate, *partial* state space exploration, similar to the use of the sweep-line method in [4,7] is still possible. Since even a poorly chosen predicate can cut some of the occurrence sequences short, using it in partial state space exploration might enable the method to explore larger parts of the state space. This, of course, will depend heavily on the model under consideration and the predicate used.

In the stop-and-wait example, a cut-off predicate could be the predicate that maps a state to $\mathtt{tt}$ if the sender sequence number is equal to the next expected sequence number at the receiver, and maps all other states to $\mathtt{ff}$. This predicate would be satisfied in all states where a new acknowledgement has just been successfully delivered to the sender. As a consequence, the predicate will be satisfied infinitely often in any progress-unbounded occurrence sequence, and thus is a cut-off predicate. In the next section we examine the choice of both cut-off predicates and progress measures in greater detail.

6 Experimental Results

A prototype for the new method has been implemented on top of the state space tool of Design/CPN [16,3,6], by combining the existing equivalence reduction facilities [14] with the sweep-line prototype described in [7]. In this section we evaluate the new method and the prototype on two case studies. The first being the stop-and-wait protocol we used for introducing the new method, the second the industrial case study described in [5], in which Timed Coloured Petri Nets and the Design/CPN tool were used to validate vital parts of the Bang and Olufsen BeoLink system. All experiments were conducted on a Pentium III 500Mhz Linux PC with 128Mbyte RAM.

Stop-and-Wait Protocol. In the previous sections we have already discussed the choice of equivalence relation (abstract away data and absolute sequence numbers), choice of progress measure (the receiver sequence number), and choice of cut-off predicate (sender sequence number equal to receiver sequence number). The comparison between the plain equivalence reduction and the new method, with these parameters, is shown in Table 1, for different values for buffer capacity for the underlying network. In the Equiv. Reduction column, the number of equivalence classes in the reduced state space is shown, together with the generation time for the reduced state space. In the Combined Method column the total number of states seen during the sweep, together with the maximum number of states stored at one time, and the number of equivalence classes stored all in all, and the runtime for the method. The sum of the maximum states stored and the number of classes stored is an upper bound on the memory usage of the method.

As it turns out, the choice of cut-off predicate seems to be rather poor, since the predicate selects a large percentage of the processed states. In addition the maximal number of states stored in the sweep exceeds half of the total number of equivalence classes. The result is that the sum of states stored in the sweep (Maximum) and the number of equivalence classes (Classes) exceed the number of equivalence classes in the equivalence reduced state space (i.e. the new method performs worse than the equivalence reduction in memory usage), and in addition the overhead in maintaining two tables, one for ordinary states and one for equivalence classes, gives the new method a much worse runtime performance than the equivalence reduction method.

However, by choosing a slightly different cut-off predicate, the performance of the new method can be improved. Instead of selecting all states in which the sequence numbers of sender and receiver are equal we can just select the states in which the sender has received an acknowledgement for the current sequence number, but has not yet started transmitting the next packet. A predicate selecting these states is one requiring that the number on NextSend at the sender is equal to the number on NextReceive at the receiver, and that the status on NextSend at the sender is acked. Using this cut-off predicate, the new method uses less memory than the equivalence reduction method, as seen in Table 2 and Fig. 9. Perhaps surprisingly the same total number of states is explored with

Table 1. Stop-and-Wait Protocol

	Equiv. Reduction		Combined Method			
Buffer Cap.	Classes	Time	Total	Maximum	Classes	Time
1	272	00:00:00	376	376	168	00:00:00
2	2,016	00:00:04	2,574	2,002	1,170	00:00:09
3	8,384	00:00:37	10,208	6,944	4,704	00:01:57
4	25,400	00:03:54	29,950	17,457	13,950	00:17:37
5	62,928	00:20:33	72,504	38,758	34,056	01:42:32

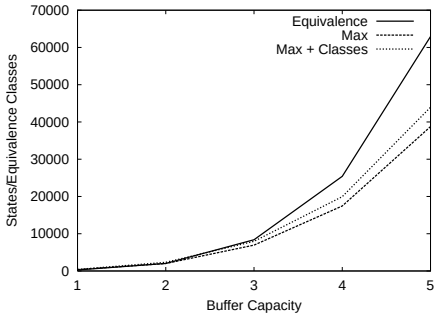
Table 2. Stop-and-Wait Communication Protocol – improved cut-off predicate

	Equiv. Reduction		Combined Method			
Buffer Cap.	Classes	Time	Total	Maximum	Classes	Time
1	272	00:00:00	376	376	64	00:00:00
2	2,016	00:00:04	2,574	2,002	324	00:00:05
3	8,384	00:00:37	10,208	6,944	1,024	00:00:47
4	25,400	00:03:54	29,950	17,457	2,500	00:06:52
5	62,928	00:20:33	72,504	38,758	5,184	00:35:01

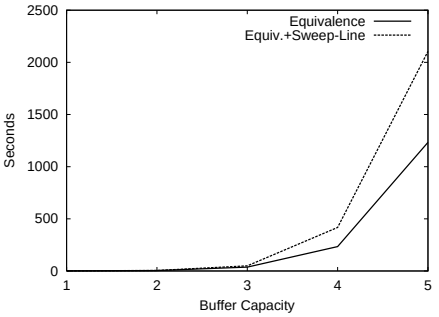
both choices of cut-off predicates. This seems to indicate that the extra classes stored using the first predicate are in fact not responsible for any of the cut-off points. The runtime performance of the new method is still worse than the runtime performance of the equivalence reduction method though.

As apparent from Fig. 9(a), storing the states with the same progress measure, i.e. the Maximum column in Table 2 and the Max line in the graph, is by far the most memory consuming part of the method on this model. The equivalence classes stored, i.e. the difference between the Max and Max + Classes lines, adds very little to the memory consumption. This observation motivates an improvement of the progress measure: If we can split up the sets of states with identical progress into smaller sets, the maximal number of states that has to be stored at the same time will be smaller, and the memory usage will be improved.

One improvement is to include the sequence numbers on packets buffered on the lower layer protocol in the definition of the progress measure. We observe that once the sender sequence number has been increased as the result of an acknowledgement, all new packets sent to the underlying protocol layer will have sequence number equal to or higher than the new sender sequence number. In other words, whenever the sender's sequence number is fixed, the number of packets buffered in the underlying layer, with sequence number less than the sender's sequence number, is monotonically decreasing. Using this observation we say that one state s has progressed less than another state s' if either the sender sequence number in s is less than it is in s' or if the sequence number is the same in s and s', but s has more packets with smaller sequence numbers buffered than has s'. Similarly the receiver sequence number and sequence numbers of



(a) Memory usage for equivalence reduction vs. the combined sweep-line and equivalence method.



(b) Runtime for equivalence reduction vs. the combined sweep-line and equivalence method

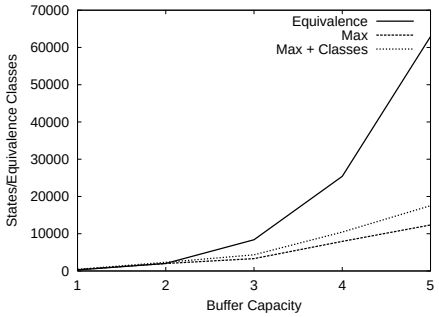
Fig. 9. Time and space usage for the protocol model

Table 3. Stop-and-Wait Protocol – improved cut-off predicate and improved progress-measure

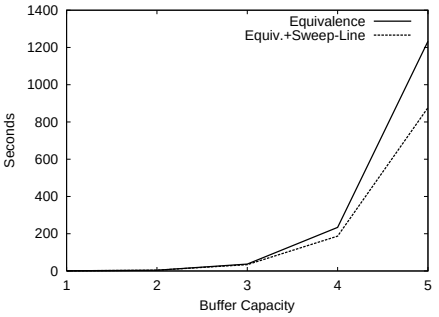
	Equiv. Reduction		Combined Method			
Buffer Cap.	Classes	Time	Total	Maximum Classes		Time
1	272	00:00:00	376	376	64	00:00:00
2	2,016	00:00:04	2,574	2,004	324	00:00:05
3	8,384	00:00:37	10,354	3,315	1,024	00:00:34
4	25,400	00:03:54	30,320	7,950	2,500	00:03:07
5	62,928	00:20:33	74,135	12,350	5,184	00:14:37

buffered acknowledgements can be take into account. Improving the progress measure in this way gives us the results shown in Table 3 and Fig. 10. Having fewer nodes to examine when testing whether a given state is already present in the set of nodes also improves the runtime to the point where the new method performs better than the equivalence reduction method, as seen in Fig. 10(b).

Bang & Olufsen BeoLink System. Our next example is the industrial case-study described in detail in [5]. Here Timed Coloured Petri Nets were used to model and analyse the Bang and Olufsen BeoLink system. The BeoLink system makes it possible to distribute audio and video throughout a home via a dedicated network. The analysis in [5] focused on the *lock management protocol* of the BeoLink system. This protocol is used to grant devices exclusive access to various services in the system. The exclusive access is implemented based on the notion of a *key*. A device is required to possess the key in order to



(a) Memory usage for equivalence reduction vs. the combined sweep-line and equivalence method.



(b) Runtime for equivalence reduction vs. the combined sweep-line and equivalence method

Fig. 10. Time and space usage for the protocol model using the improved progress measure

access services. When the system boots no key exists, and the lock management protocol is (among other things) responsible for ensuring that a key is generated when the system starts. It is the obligation of the so-called *video* or *audio master* device to ensure that new keys are generated when needed. In [5] partial state space analysis was used to analyse the boot-phase of the system, and in [7] the sweep-line method was used to analyse the boot-phase for larger configurations of the system. In [8] equivalence reduction was used to explore the (reduced) state space of the entire system, i.e. not only the boot-phase but also the behaviour following boot-up.

We combine equivalence reduction and the sweep-line method by choosing the so-called *termination-time equivalence* from [8] as the equivalence relation and the global time from Timed Coloured Petri Nets as our progress measure. This reflects the approaches taken in [8] and [7]. The lock management protocol must ensure the creation of a key if a key does not already exist, so for our first choice of cut-off predicate we use the existence of a key. The performance of the method with these parameters is shown in Table 4. Configurations with one video master are written as VM:n, where n is the total number of devices in the system. Configurations with one audio master are written as AM:n.

As seen in Table 4, the maximal number of states needed for the sweep is small compared to the number of states explored in total and also small compared to the total number of equivalence classes. However, the number of equivalence classes saved in the combined method is very high, almost as high as the total number of equivalence classes in the equivalence reduced state space, which indicates that the cut-off predicate chosen is accepting too many states. As a more restrictive predicate we can choose the predicate that is true whenever

Table 4. Bang & Olufsen BeoLink System – Key exists predicate

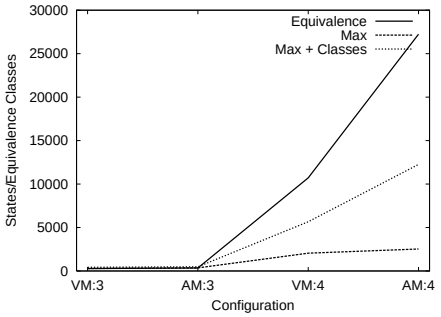
	Equiv. Reduction		Combined Method			
Config.	Classes	Time	Total	Maximum Classes		Time
VM:3	274	00:00:01	275	275	228	00:00:01
AM:3	346	00:00:01	348	348	268	00:00:01
VM:4	10,713	00:01:27	10,779	2,136	8,902	00:01:13
AM:4	27,240	00:04:12	27,400	2,531	22,424	00:03:32

Table 5. Bang & Olufsen BeoLink System – Key not in use predicate

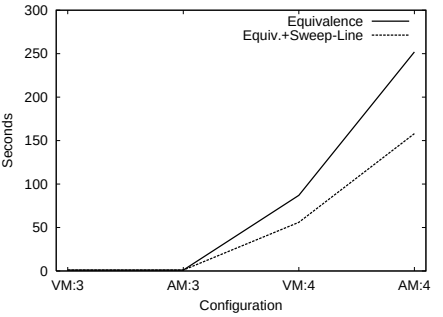
	Equiv. Reduction		Combined Method			
Config.	Classes	Time	Total	Maximum Classes		Time
VM:2	274	00:00:01	284	284	136	00:00:01
AM:2	346	00:00:01	354	354	127	00:00:01
VM:3	10,713	00:01:27	11,321	2,047	3,616	00:00:56
AM:3	27,240	00:04:12	28,514	2,531	9,728	00:02:38

the key exists but is not currently in use. This predicate will be true whenever a component has released the key but the next component has not acquired it yet. Using this predicate we get the results shown in Table 5 and Fig. 11.

This last example is particularly interesting because it shows that the increasing global time of the TCPN formalism can be used to *improve* the performance of state space analysis, and is not just an inconvenience to get rid of as soon as possible. The approach taken in both Timed Coloured Petri Nets [12] and Interval Timed Coloured Petri Nets [17] for modelling time is to have a *global clock* modelling the current time, and to extend tokens with *time stamps* specifying when the tokens are *available* for consumption. The execution of nets is time driven in the sense that whenever no transition is enabled, due to tokens being unavailable, the global clock is increased to the earliest time at which a transition is enabled. While intuitively simple, this approach has the drawback that many models have an infinite number of states, due to the increasing time values. This problem has to a large extent been solved by grouping states into equivalence classes in [1,8], which, in effect, replaces the "increasing global time" semantics with a "delay" semantics: Instead of giving tokens an (absolute) time stamp value, and making the tokens unavailable until the global clock has reached this value, the tokens are delayed for a period after their creation. Rather than using increasing absolute time values in the formalism, and translating this into delays in the analysis, one could also take the approach of using delays in the formalism in the first place. The experimental results presented above, however, argues against this. Replacing the increasing time values with delays would result in a state space equal to the equivalence reduced state space, and we would not be able to use time-progress for the more efficient combined sweep-line and equivalence reduction method.



(a) Memory usage for equivalence reduction vs. the combined sweep-line and equivalence method



(b) Runtime for equivalence reduction vs. the combined sweep-line and equivalence method

Fig. 11. Time and space usage for the BeoLink system

7 Conclusion

We have presented a new state space exploration method combining the sweep-line method with equivalence reduction for analysis of a certain class of infinite state systems. Equivalence reduction is used to obtain a finite number of states, while the sweep-line method is used for exploring the state space until all reachable equivalence classes have been visited at least once. The aim of the method is to reduce memory consumption compared to equivalence reduction, without having to settle for partial state space exploration as with the sweep-line method alone.

We have developed tool support for the method on top of the state space tool of Design/CPN, and evaluated the new method on two examples, one of which is an industrial case-study. The experimental results show that the new method can indeed result in better performance than equivalence reduction, but the experiments also show that the performance of the method is very sensitive to the choice of both progress measure and cut-off predicate. Our experiments show performance ranging from a slight space reduction, at the cost of a severe runtime penalty, to significant reductions in both space and time, depending on the choice of progress measure and predicate. Not only can a poorly chosen predicate impair performance, if the predicate is not a cut-off predicate, in the formal sense defined in Sect. 5, the state space exploration is not guaranteed to terminate, and we may have to rely on partial state space exploration. Methods for automatically choosing good progress measures and cut-off predicates would therefore be of great value. This is left for future work.

Acknowledgements

The author would like to thank Lars M. Kristensen and Charles Lakos for their comments on earlier versions of this paper.

References

1. G. Berthelot. Occurrence Graphs for Interval Timed Coloured Nets. In *Proceedings of ICATPN'94*, volume 815 of *Lecture Notes in Computer Science*, pages 79–98. Springer Verlag, 1994. 315, 331
2. D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., 1992. 316
3. S. Christensen, K. Jensen, and L.M. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark. Online version: `http://www.daimi.au.dk/designCPN/`. 327
4. S. Christensen, K. Jensen, T. Mailund, and L. M. Kristensen. State Space Methods for Timed Coloured Petri Nets. In *Proceedings of 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin Germany, September 2001. 315, 316, 326
5. S. Christensen and J. B. Jørgensen. Analysis of Bang and Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets. In P. Azéma and G. Balbo, editors, *Proceedings of ICATPN'97*, volume 1248 of *Lecture Notes in Computer Science*, pages 387–406. Springer-Verlag, 1997. 327, 329, 330
6. S. Christensen, J. B. Jørgensen, and L. M. Kristensen. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 1997. 327
7. S. Christensen, L. M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In Tiziana Margaria and Wang Yi, editors, *Proceedings of TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001. 315, 317, 318, 319, 326, 327, 330
8. S. Christensen, L.M. Kristensen, and T. Mailund. Condensed State Spaces for Timed Petri Nets. In José-Manuel Colom and Maciej Koutny, editors, *Proceedings of ICATPN'01*, volume 2075 of *Lecture Notes in Computer Science*, pages 101–120. Springer-Verlag, 2001. 315, 330, 331
9. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. *Formal Methods in System Design*, 9, 1996. 320
10. E. A. Emerson and A. P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9, 1996. 320
11. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 1: Basic Concepts*. Monographs in theoretical computer science, Springer-Verlag, Berlin, 1992. 316
12. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 2: Analysis Methods*. Monographs in theoretical computer science, Springer-Verlag, Berlin, 1994. 315, 319, 320, 322, 331
13. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9, 1996. 320
14. J. B. Jørgensen and L. M. Kristensen. *Design/CPN Condensed State Space Tool Manual*. Department of Computer Science, University of Aarhus, Denmark, 1996. Online: `http://www.daimi.au.dk/designCPN/`. 327

15. L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, December 1998. 315, 316
16. Design/CPN Online. `http://www.daimi.au.dk/designCPN/`. 327
17. W. M. P. van der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In *Proceedings of ICATPN'93*, volume 691 of *Lecture Notes in Computer Science*, pages 453–472. Springer Verlag, 1993. 315, 331

Regular Event Structures and Finite Petri Nets: The Conflict-Free Case*

Mogens Nielsen[1] and P. S. Thiagarajan[2]**

[1] BRICS***, Computer Science Department
University of Aarhus, Denmark
[2] School of Computing
National University of Singapore, Singapore

Abstract. We present the notion of regular event structures and conjecture that they correspond exactly to finite 1-safe Petri nets. We show that the conjecture holds for the conflict-free case. Even in this restricted setting, the proof is non-trivial and involves a natural subclass of regular event structures that admit a sensible labeling with Mazurkiewicz trace alphabets.

1 Introduction

A classic result in concurrency theory is that the non-interleaved branching time behavior of 1-safe Petri nets can be represented as prime event structures [NPW]. Since then, this relationship between Petri nets and event structures has been strengthened in various ways [NRT, WN]; so much so, these two classes of objects can now be viewed as being strongly equivalent.

Here we study a conjecture that relates Petri nets and event structures in the presence of a *finiteness* assumption; the property of event structures that we call *regular* and which we conjecture corresponds to a 1-safe Petri net being finite [Thi2].

Finite 1-safe Petri nets are important because they constitute a basic model of finite state distributed systems. Indeed, a good deal of research has been done on succinctly representing the event structure unfolding of a 1-safe Petri net (for a sample of the literature, see [Mc, E, ERV]). Here, the simple but basic observation due to Ken McMillan [Mc] is that if the 1-safe Petri net is finite, then its potentially infinite event structure unfolding can be represented as a *finite* object. The conjecture we formulate places this observation in a more general setting.

More precisely, we define the notion of *regular* event structures and conjecture that finite 1-safe Petri nets and regular event structures correspond to each other. It turns out that this is a smooth generalization of the classical case,

* A major part of this work was done at BRICS
** On leave from Chennai Mathematical Institute, Chennai, India
*** Basic Research In Computer Science, a centre funded by the Danish National Research Foundation

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 335–351, 2002.
© Springer-Verlag Berlin Heidelberg 2002

where it is folklore that finite transition systems and regular trees correspond to each other. It is worth noting that the notion of a regular tree is fundamental in the setting of branching time temporal and modal logics [Tho]. We expect regular event structures to play a similar role in non-interleaved branching time settings. Admittedly, we still lack at present an effective temporal/modal logic over labeled event structures and hence at this stage we mainly wish to draw attention to our conjecture in terms of its naturality.

We also wish to add that a proof of our conjecture could be useful in situations involving the folding of an event structure into a finite Petri net. One such example is the proof of the undecidability of hereditary history-preserving bisimilarity for finite one-safe nets from [JN]. This proof is based on a reduction to the bisimilarity of certain regular event structures, and a substantial part of the proof is concerned with establishing that these event structures are indeed unfoldings of finite one-safe nets, a fact which would follow immediately from our conjecture. A second example is [MT] where the undecidability of the controller synthesis problem is established in a number of distributed settings. Here again, ad hoc arguments are used to show that certain regular event structures fold down to restricted finite 1-safe Petri nets represented as distributed transition systems.

In the next section we define regular event structures and formulate the conjecture. In section 3, we show that the subclass of conflict-free regular event structures correspond to conflict-free 1-safe Petri nets. The two components of this proof are: Firstly, regular conflict-free event structures admit a sensible labeling in terms of Mazurkiewicz trace alphabets. Secondly, such labeled conflict-free regular event structures correspond to finite conflict-free 1-safe Petri nets. In section 4 we show that the folklore result involving finite transition systems and regular trees leads more or less directly to the fact that the conjecture holds also for the sequential case, the setting where there is no concurrency.

2 The Conjecture

We start with some notations concerning posets.

Let $(X, \leq)$ be a poset and $Y \subseteq X$. Then $\downarrow Y = \{x \mid \exists y \in Y, x \leq y\}$ and $\uparrow Y = \{x \mid \exists y \in Y, x \geq y\}$. Whenever Y is a singleton with $Y = \{y\}$ we will write $\downarrow y$ ($\uparrow y$) instead of $\downarrow\{y\}$ ($\uparrow\{y\}$). $\lessdot \subseteq X \times X$ is defined as: $x \lessdot y$ iff $x < y$ (i.e. $x \leq y$ and $x \neq y$) and for every z, if $x \leq z \leq y$, then $x = z$ or $z = y$. $\lessdot$ is also used as an operator on subsets of X in the standard way: $\lessdot Y = \{x \mid \exists y \in Y, y \lessdot x\}$. In case $Y = \{y\}$ is a singleton, we will write $\lessdot y$ instead of $\lessdot Y$.

A *prime event structure* is a triple $ES = (E, \leq, \#)$ where $(E, \leq)$ is a poset and $\# \subseteq E \times E$ is an irreflexive and symmetric relation such that the following conditions are met:

- $\downarrow e$ is a finite set for every $e \in E$,
- for every $e_1, e_2, e_3 \in E$, if $e_1 \# e_2$ and $e_2 \leq e_3$ then $e_1 \# e_3$.

E is the set of *events*, $\leq$ is the *causality* relation, and $\#$ is the *conflict* relation. Events not related by $\leq$, $\geq$ or $\#$ are said to be concurrent, $co = E \times E \backslash (\leq \cup \geq$

$\cup \#$). Throughout what follows, we shall refer to prime event structures as just event structures.

As usual, the states of an event structure will be called configurations. Let $ES = (E, \leq, \#)$ be an event structure. We say that $c \subseteq E$ is a configuration iff $c = \downarrow c$ and $(c \times c) \cap \# = \emptyset$. It is easy to see that $\emptyset$ is always a configuration and $\downarrow e$ is a configuration for every event e. We let C_{ES} denote the set of finite configurations of ES.

Let $c \in C_{ES}$. We define $\#(c) = \{e' \mid \exists e \in c.\ e \ \# \ e'\}$. The substructure rooted at c is denoted by $ES\backslash c$ and is defined to be the triple $ES\backslash c = (E', \leq', \#')$ where

- $E' = E - (c \cup \#(c))$,
- $\leq'$ is $\leq$ restricted to $E' \times E'$,
- $\#'$ is $\#$ restricted to $E' \times E'$.

Clearly, $ES\backslash c$ is also an event structure.

Let $ES_i = (E_i, \leq_i, \#_i)$, $i = 1, 2$ be a pair of event structures. We say that $f : E_1 \to E_2$ is an isomorphism between ES_1 and ES_2 iff f is a bijection such that $e_1 \leq_1 e_1'$ iff $f(e_1) \leq_2 f(e_1')$ and $e_1 \ \#_1 \ e_1'$ iff $f(e_1) \ \#_2 \ f(e_1')$ for every $e_1, e_1' \in E_1$. If such an isomorphism exists, we say that ES_1 and ES_2 are isomorphic – and denote this by $ES_1 \equiv ES_2$.

Finally, for the event structure $ES = (E, \leq, \#)$ we define the equivalence relation $R_{ES} \subseteq C_{ES} \times C_{ES}$ via:

$$c \ R_{ES} \ c' \text{ iff } ES\backslash c \equiv ES\backslash c'.$$

The set of events enabled at a configuration will also play a role in the definition of a regular event structure. To this end, let $ES = (E, \leq, \#)$ be an event structure. Then we say that $e \in E$ is enabled at $c \in C_{ES}$ iff $e \notin c$ and $c \cup \{e\} \in C_{ES}$. Let $en(c)$ be the set of events enabled at the configuration c.

Definition 1. *The event structure ES is* regular *iff R_{ES} is of finite index and there exists an integer d such that $|en(c)| \leq d$ for every $c \in C_{ES}$.*

Intuitively, the transition system associated with ES is required to be a boundedly-branching dag with finite number of isomorphism classes; thus generalizing the standard notion of a regular tree [Tho]. It is not difficult to verify that the condition requiring the boundedly-branching property is independent of R_{ES} being required to be of finite index.

We next define a 1-safe Petri net to be a quadruple $\mathcal{N} = (S, T, F, M_{in})$ where (S, T, F) is a net and $M_{in} \subseteq S$ is the initial marking. We say that $\mathcal{N}$ is finite iff both S and T are finite sets.

The basic result shown in [NPW] says that every 1-safe Petri net unfolds into an event structure. We shall begin by recalling how one associates an event structure with a 1-safe Petri net.

Fix $\mathcal{N} = (S, T, F, M_{in})$, a 1-safe Petri net. As usual, for $x \in S \cup T$, we set ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. The dynamics of $\mathcal{N}$ are captured by the associated transition system $TS_{\mathcal{N}} = (RM_{\mathcal{N}}, \to_{\mathcal{N}}, M_{in})$ where $RM_{\mathcal{N}} \subseteq 2^S$ and $\to_{\mathcal{N}} \subseteq RM_{\mathcal{N}} \times T \times RM_{\mathcal{N}}$ are the least sets satisfying:

- $M_{in} \in RM_{\mathcal{N}}$,
- suppose $M \in RM_{\mathcal{N}}$ and $t \in T$ such that ${}^\bullet t \subseteq M$ and $(t^\bullet - {}^\bullet t) \cap M = \emptyset$, then $M' \in RM_{\mathcal{N}}$ and $M \xrightarrow{t}_{\mathcal{N}} M'$ where $M' = (M - {}^\bullet t) \cup t^\bullet$.

We next extend the relation $\rightarrow_{\mathcal{N}}$ to sequences of transitions. This extension is denoted by $\Longrightarrow_{\mathcal{N}} \subseteq RM_{\mathcal{N}} \times T^\star \times RM_{\mathcal{N}}$, and is defined via:

- $M \overset{\epsilon}{\Longrightarrow}_{\mathcal{N}} M$ for every M in $RM_{\mathcal{N}}$,
- suppose $M \overset{\sigma}{\Longrightarrow}_{\mathcal{N}} M'$ with $\sigma \in T^\star$ and $M' \xrightarrow{t}_{\mathcal{N}} M''$ then $M \overset{\sigma t}{\Longrightarrow}_{\mathcal{N}} M''$.

This leads to the notion of the set of firing sequences of $\mathcal{N}$ denoted $FS_{\mathcal{N}}$ which is given by:

$\sigma \in FS_{\mathcal{N}}$ iff there exists $M \in RM_{\mathcal{N}}$ such that $M_{in} \overset{\sigma}{\Longrightarrow}_{\mathcal{N}} M$.

We will extract the event structure unfolding of $\mathcal{N}$ from $FS_{\mathcal{N}}$ using the theory of Mazurkiewicz traces. For basic material concerning this rich theory, which we shall assume, the reader is referred to [DR]. Here we wish to merely recall that a (Mazurkiewicz) trace alphabet is a pair $M = (\Sigma, I)$, where Σ is a finite non-empty alphabet set and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric relation called the independence relation. D is the dependence relation given by $D = (\Sigma \times \Sigma) - I$. Further, $\sim_I$ is the least equivalence relation induced by I over $\Sigma^\star$ satisfying: $\sigma_1 ab \sigma_2 \sim_I \sigma_1 ba \sigma_2$ whenever $\sigma_1, \sigma_2 \in \Sigma^\star$ and $(a, b) \in I$.

Returning to $\mathcal{N}$, we first note that $(T, I_{\mathcal{N}})$ is a trace alphabet, where $I_{\mathcal{N}}$ is given by: $t_1 \; I_{\mathcal{N}} \; t_2$ iff $({}^\bullet t_1 \cup t_1^\bullet) \cap ({}^\bullet t_2 \cup t_2^\bullet) = \emptyset$.

We let $\sim_{\mathcal{N}}$ denote the equivalence (congruence) induced by $I_{\mathcal{N}}$ over $T^\star$. For σ in $T^\star$, we let $\langle \sigma \rangle$ denote the $\sim_{\mathcal{N}}$-equivalence class (trace) containing σ.

A simple but crucial fact is that in case $\mathcal{N}$ is finite, $FS_{\mathcal{N}}$ is a prefix-closed regular subset of $T^\star$. A second important fact is that $FS_{\mathcal{N}}$ is trace-closed. In other words, if $\sigma \in FS_{\mathcal{N}}$ then $\langle \sigma \rangle \subseteq FS_{\mathcal{N}}$.

We next define the relation $\sqsubseteq$ over $T^\star / \sim_{\mathcal{N}}$ as follows:

$\langle \sigma \rangle \sqsubseteq \langle \sigma' \rangle$ iff there exists σ_1 in $\langle \sigma \rangle$ and σ_1' in $\langle \sigma' \rangle$ such that σ_1 is a prefix of σ_1'.

Finally, for σ in $T^\star \backslash \{\epsilon\}$, we let $last(\sigma)$ be the last letter appearing in σ. We define $\langle \sigma \rangle$ to be a *prime* trace iff σ is a non-null sequence and for every σ' in $\langle \sigma \rangle$, it is the case that $last(\sigma) = last(\sigma')$. Thus for a prime trace $\langle \sigma \rangle$, it makes sense to define $last(\langle \sigma \rangle)$ via: $last(\langle \sigma \rangle) = last(\sigma)$.

At last, we can define $ES_{\mathcal{N}} = (E, \leq, \#)$ where:

- $E = \{\langle \sigma \rangle | \sigma \in FS_{\mathcal{N}} \text{ and } \langle \sigma \rangle \text{ is prime}\}$,
- $\leq$ is $\sqsubseteq$ restricted to $E \times E$,
- for all $e, e' \in E, e \# e'$ iff there does *not* exist $\sigma \in FS_{\mathcal{N}}$ such that $e \sqsubseteq \langle \sigma \rangle$ and $e' \sqsubseteq \langle \sigma \rangle$.

It is easy to check that $ES_{\mathcal{N}}$ is a prime event structure. We have now arrived at the main point of the paper :

CONJECTURE

The event structure ES is regular iff there exists a *finite* 1-safe Petri net $\mathcal{N}$ such that ES and $ES_{\mathcal{N}}$ are isomorphic.

Thus our conjecture says that for 1-safe Petri nets, finiteness at the net level corresponds to regularity at the event structure level. One half of the conjecture is easy to show; namely $ES_{\mathcal{N}}$ is regular for every finite 1-safe Petri net $\mathcal{N}$. This result follows from a series of standard definitions and observations which we shall now sketch.

It turns out that there is a close relationship between the dynamics of $\mathcal{N}$ and $ES_{\mathcal{N}}$. This can be brought out through the map *Config*, which assigns a configuration of $ES_{\mathcal{N}}$ to each firing sequence of $\mathcal{N}$, and the map Tr which assigns a trace (contained in $FS_{\mathcal{N}}$) to each configuration of $ES_{\mathcal{N}}$. Both these maps can be defined inductively as follows:

- $\mathit{Config}(\epsilon) = \emptyset$ and $Tr(\emptyset) = \{\epsilon\}$.
- Suppose σt is in $FS_{\mathcal{N}}$ with $\mathit{Config}(\sigma) = c$. Then $\mathit{Config}(\sigma t) = c \cup \{e\}$ where e is the unique event enabled at c satisfying $\mathit{last}(e) = t$. Next suppose $c' = c \cup \{e\}$ is a configuration such that e is enabled at c and $Tr(c) = \langle\sigma\rangle$. Then $Tr(c') = \langle\sigma t\rangle$ where $\mathit{last}(e) = t$.

It is not difficult to verify that the maps *Config* and Tr are well-defined. One can in fact show that the map *Config* can be consistently extended to $FS_{\mathcal{N}}/\sim_{\mathcal{N}}$ via: $\mathit{Config}(\langle\sigma\rangle) = \mathit{Config}(\sigma)$. Further, one can now associate a reachable marking of $\mathcal{N}$ with each configuration of $ES_{\mathcal{N}}$ with the help of the map *Mark* given by:

$$\mathit{Mark}(c) = M \text{ iff } M_{in} \stackrel{\sigma}{\Longrightarrow}_{\mathcal{N}} M \text{ where } Tr(c) = \langle\sigma\rangle.$$

The point is, it is straightforward to show that the event structure rooted at c ($ES_{\mathcal{N}}\backslash c$) is isomorphic to the event structure unfolding of the 1-safe Petri net whose underlying net is that of $\mathcal{N}$ but whose initial marking is $\mathit{Mark}(c)$. Thus the equivalence relation $R_{\mathcal{N}}$, which sets two configurations c and c' to be equivalent just in case $\mathit{Mark}(c) = \mathit{Mark}(c')$, will be of finite index *and* it will be a refinement of the equivalence relation $R_{ES_{\mathcal{N}}}$. Consequently, $R_{ES_{\mathcal{N}}}$ is of finite index as well. Further, at most $|T|$ events are enabled at any configuration of $ES_{\mathcal{N}}$. Thus $ES_{\mathcal{N}}$ is regular, and therefore, if the event structure ES is isomorphic to $ES_{\mathcal{N}}$, then ES too is regular.

The second half of the conjecture turns out to be surprisingly difficult to show. More precisely, we do not know how to do it. One source of difficulty is that there could be more than one isomorphism relating two configurations (i.e. the sub-event structures rooted at two configurations) and there seems to be no good way of fixing these isomorphisms globally and uniformly.

From the results of [Thi1], where the notion of regular *trace event structures* was introduced, it follows that our conjecture holds for this subclass of event structures. Trace event structures are the underlying event structures of certain kinds of labeled event structures, where events are labeled with the letters of a Mazurkiewicz trace alphabet. The pertinent results of [Thi1] involve the use of a variety of technical notions and hence we do not wish to appeal to these results directly. We will however establish independently the result, that *in the conflict-free case*, every regular trace event structure is isomorphic to the event structure

unfolding of a finite 1-safe Petri net. As preparation, it will be convenient to recall the notion of regular trace event structures.

In our formulation we shall make use of the following derived conflict relation. Let $ES = (E, \leq, \#)$ be an event structure. The minimal conflict relation $\#_\mu \subseteq E \times E$ is defined via:

$$e \ \#_\mu \ e' \text{ iff } (\downarrow e \times \downarrow e') \cap \# = \{(e, e')\}.$$

Definition 2. *Let $M = (\Sigma, I)$ be a trace alphabet. An M-labeled event structure is a structure $LES = (ES, \lambda)$ where $ES = (E, \leq, \#)$ is an event structure and $\lambda : E \to \Sigma$ is a labeling function which satisfies :*

(LES1) $e \ \#_\mu \ e'$ *implies* $\lambda(e) \neq \lambda(e')$

(LES2) *If* $e \lessdot e'$ *or* $e \ \#_\mu \ e'$ *then* $(\lambda(e), \lambda(e')) \in D$

(LES3) *If* $(\lambda(e), \lambda(e')) \in D$ *then* $e \leq e'$ *or* $e' \leq e$ *or* $e \ \# \ e'$.

The restrictions $(LES2)$ and $(LES3)$ on the labeling function ensure that the concurrency relation associated with an event structure respects the independence relation of M. The restriction $(LES1)$ demands that the natural Σ-labeled transition system associated with LES is deterministic.

In what follows we will often represent the M-labeled event structure $LES = (ES, \lambda)$ with $ES = (E, \leq, \#)$ as $LES = (E, \leq, \#, \lambda)$. We will also say that ES is the *underlying event structure* of LES.

Let $LES = (E, \leq, \#, \lambda)$ be an M-labeled event structure and $c \in C_{ES}$. As before, $\#(c) = \{e' \mid \exists e \in c.\ e \ \# \ e'\}$. We denote the substructure rooted at c as $LES\backslash c$, and define it in the obvious way. Again, it is an easy observation that $LES\backslash c$ is also an M-labeled event structure.

Let $LES_i = (E_i, \leq_i, \#_i, \lambda_i)$, $i = 1, 2$ be a pair of M-labeled event structures. We say that LES_1 and LES_2 are isomorphic – and again denote this by $LES_1 \equiv LES_2$ – iff there exists an isomorphism $f : E_1 \to E_2$ between ES_1 and ES_2, such that $\lambda_2(f(e_1)) = \lambda_1(e_1)$ for every $e_1 \in E_1$.

Finally, for the M-labeled event structure $LES = (ES, \lambda)$, the equivalence relation $R_{LES} \subseteq C_{ES} \times C_{ES}$ is given by:

$$c \ R_{LES} \ c' \text{ iff } LES\backslash c \equiv LES\backslash c'.$$

We can now define regular trace event structures in two steps.

Definition 3. *The M-labeled event structure LES is regular iff R_{LES} is of finite index.*

Unlike the case of (unlabeled) event structures we do not have to demand here that at every configuration only a bounded number of events are enabled. Due to the conditions $LES1$ and $LES3$ in the definition of an M-labeled event structure, we are guaranteed that at each configuration at most $|\Sigma|$ events are enabled.

Definition 4. *The event structure ES is a regular* trace *event structure iff there exists a trace alphabet M and a regular M-labeled event structure LES such that ES is isomorphic to the underlying event structure of LES.*

It is clear from the definition that every regular trace event structure is also a regular event structure. In fact, it is not difficult to show that our conjecture is equivalent to the assertion that every regular event structure is also a regular trace event structure. As mentioned earlier, we do not know how to prove this assertion in general. The bulk of the next section will be devoted to showing that the assertion indeed holds for conflict-free event structures.

3 The Conflict-Free Case

A conflict-free event structure is an event structure with empty conflict relation. Such an event structure can be viewed as just a partially ordered set of events, and hence in the following we sometimes denote a conflict-free event structure simply by $(E, \leq)$. To state the main result of the paper we also need the notion of a (behaviorally) conflict-free 1-safe Petri net. Let $\mathcal{N} = (S, T, F, M_{in})$ be a 1-safe Petri net. Let M be a reachable marking of $\mathcal{N}$. Then $t \in T$ is enabled at M iff there exists M' such that $M \xrightarrow{t}_{\mathcal{N}} M'$. We say that $\mathcal{N}$ is *behaviorally conflict-free* (from now we will just say *conflict-free*) iff for every reachable marking M, if t_1 and t_2 are enabled at M with $t_1 \neq t_2$, then the F-neighborhoods of t_1 and t_2 are disjoint. In other words, $t_1 \; I_{\mathcal{N}} \; t_2$. The main result of the paper is :

Theorem 1. *Let ES be an event structure. Then ES is a regular conflict-free event structure iff it is isomorphic to the event structure unfolding of a finite conflict-free 1-safe Petri net.*

As argued in Section 2, the event structure unfolding of a finite 1-safe Petri net is regular. From this it follows easily that the event structure unfolding of a finite conflict-free 1-safe Petri net is a regular conflict-free event structure. The converse however does not follow easily from known facts. Its proof is split into two major steps. First, we show that a regular conflict-free event structure is also a regular conflict-free trace event structure (Theorem 2). As a second step, we show that every regular conflict-free trace event structure is isomorphic to the event structure unfolding of a finite conflict-free 1-safe Petri net.

We start by showing a useful structural property of regular partial orders.

Lemma 1. *Let $ES = (E, \leq)$ be a regular conflict-free event structure. Then there exists an integer k such that for all $e \in E$*

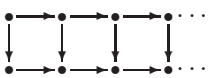
$$|\downarrow(\uparrow e) \setminus (\uparrow e \cup \downarrow e)| \quad < \quad k$$

Proof. Let $e \in E$. It will be convenient to denote $\downarrow(\uparrow e) \setminus (\uparrow e \cup \downarrow e)$ as co-cone(e), and to define $c_e = (\downarrow e) \setminus \{e\}$. Clearly c_e is a finite configuration and e is minimal in $ES \setminus c_e$. A further easy observation is that co-cone(e) in ES is the *same set* as co-cone(e) in $ES \setminus c_e$. Thus the values of $|$co-cone$(e)|$ as e ranges

over E are determined by the values of $|\text{co-cone}(\widehat{e})|$ for the finitely many minimal elements $\widehat{e}$ in finitely many event structures - one for each R_{ES}-equivalence class of configurations of ES. Hence we only need to argue that for any minimal element e in each of these (regular conflict-free) event structures, $|\text{co-cone}(e)|$ is finite.

Suppose e is a minimal element in one such regular conflict-free event structure. We will say that $span(e)$ in this event structure is the maximum value attained by $|{\downarrow}e'|$ as e' ranges over the set $\lessdot\, e$. Clearly $span(e)$ is finite due to the boundedly-branching property of regular event structures. Let $\widehat{j}$ be the maximum of all such $span(e)$ values taken across the finitely many regular event structures induced by the R_{ES}-equivalence classes of ES. Now suppose that e is a minimal element in one such event structure and $|\text{co-cone}(e)|$ is not finite. Then for $j > \widehat{j}$ we can find e_0, e_1 such that $e \lessdot^j e_0$ and $e_1 \lessdot e_0$ and $e_1 \notin \uparrow e$. But this would imply that $span(e_1) > \widehat{j}$ in the event structure $ES\backslash c_{e_1}$ in which e_1 is a minimal element, a contradiction. □

Example 1. The following is an example of a partial order not satisfying the structural property of the previous lemma, and hence not a regular partial order.



□

Our next lemma states the existence of a particular kind of isomorphism for a given regular conflict-free event structure.

Lemma 2. *Let $ES = (E, \leq)$ be a regular conflict-free event structure such that E is infinite. Then there exist $c_0, c_1 \in C_{ES}$, and isomorphism g: $ES\backslash c_0 \rightarrow ES\backslash c_1$, such that*

- $c_o \subset {\downarrow}(\lessdot\, (c_0)) \subseteq c_1$,
- $e < g(e)$ *for every* $e \in ES\backslash c_0$.

Proof. First define inductively the following sequence of subsets of events:

- $Y_0 =$ the set of minimal events of ES.
- $Y_{i+1} = {\downarrow}(\lessdot\, Y_i)$.

Since E is an infinite set, it follows that $\{Y_i\}$ is a strictly increasing sequence of finite configurations. Further, $\bigcup\{Y_i \mid 0 \leq i\} = E$. Hence there exist indices $i_0 < i_1$ such that $ES\backslash Y_{i_0}$ and $ES\backslash Y_{i_1}$ are isomorphic, say with isomorphism $h : ES\backslash Y_{i_0} \rightarrow ES\backslash Y_{i_1}$.

Let h^n denote the n-fold iteration of h. From the construction of the sequence $\{Y_i\}$, and the fact that h^n is an isomorphism, it follows that for all $n > 0$ and for all e in $ES\backslash Y_{i_0}$, $h^n({\downarrow}e)$ is a proper subset of ${\downarrow}h^n(e)$. (Here ${\downarrow}h^n(e)$ is also taken to be relative to $ES\backslash Y_{i_0}$). In other words, $h^n({\downarrow}e) \subset {\downarrow}h^n(e)$. Now, assume that $h^n(e) \leq e$ for some e in $ES\backslash Y_{i_0}$. This clearly implies that ${\downarrow}h^n(e) \subseteq {\downarrow}e$, which in turn implies $h^n({\downarrow}e) \subset {\downarrow}e$, contradicting the fact that h^n is an isomorphism. Hence for all $e \in ES\backslash Y_{i_0}$ and for all $n > 0$, $e < h^n(e)$ or $e \; co \; h^n(e)$.

Now recall that regularity implies that there exists an integer d such that $|en(c)| \leq d$ for every $c \in C_{ES}$. This in turn implies that any set of pairwise *co*-related events has cardinality at most d.

So, consider for $e \in ES\backslash Y_{i_0}$, the set $\{h^n(e) \mid 0 \leq n \leq d\}$. From our observation above we must have indices $0 \leq m < n \leq d$ such that $h^m(e) < h^n(e)$. But h is an isomorphism and hence $e < h^{n-m}(e)$. It is easy to show by simple induction that if $e < h^{n-m}(e)$ then $e < h^{l(n-m)}(e)$ for all $l \geq 1$. From this we conclude that for all $e \in ES\backslash Y_{i_0}$, $e < h^{d!}(e)$ (where, as usual, $d!$ is the factorial of the integer d).

Now, the lemma follows for $c_0 = Y_{i_0}$ and $c_1 = Y_{i_0} \cup ((E\backslash Y_{i_0}) \setminus h^{d!}(E\backslash Y_{i_0}))$, and $g = h^{d!}$. □

In the following we fix k as defined in lemma 1 and c_0, c_1 and $g : ES\backslash c_0 \to ES\backslash c_1$ as defined in lemma 2. Before we state and prove our main theorem, we need a useful combined property of the constant k and the isomorphism g.

Definition 5. *Let $\lessdot_0$ and $\gtrdot_0$ denote the restrictions of $\lessdot$ and $\gtrdot$ to $E\backslash c_0$. Define R_0 as the equivalence relation on $E\backslash c_0$ as*

$$e_0 \; R_0 \; e_1 \quad \textit{iff} \quad e_0 \; (\lessdot_0 \cup \; \gtrdot_0)^\star \; e_1.$$

The relation R_0 captures "reachability" within $E\backslash c_0$, and it partitions $E\backslash c_0$ into finitely many equivalence classes.

Lemma 3. *Let $ES = (E, \leq)$ be a regular conflict-free event structure such that E is infinite. Let $k \in N$ and the isomorphism $g : ES\backslash c_0 \to ES\backslash c_1$ be as fixed above. Then*

- *for all $e_0, e_1 \in ES\backslash c_0$.* $[e_0 \lessdot e_1 \quad \Rightarrow \quad e_1 < g^{k+1}(e_0)]$,
- *for all $e_0, e_1 \in ES\backslash c_0$.* $[e_0 \; (co \cap R_0) \; e_1 \quad \Rightarrow \quad e_1 < g^{k+1}(e_0)]$.

Proof. First assume $e_0 \lessdot e_1$. From Lemma 2, $e_1 < g^k(e_1)$ and since g is an isomorphism, $g^k(e_0) < g^k(e_1)$. Thus for $0 < i \leq k$, $g^i(e_0) \in \downarrow(\uparrow e_1)$. From Lemma 1, there exists an i such that $0 < i \leq k$ and $g^i(e_0) \in \uparrow e_1 \cup \downarrow e_1$. But $g^i(e_0) \in \downarrow e_1$ contradicts $e_0 \lessdot e_1$, and hence there exists an i such that $0 < i \leq k$ and $e_1 \leq g^i(e_0) < g^{k+1}(e_0)$.

In order to show the second claim we first show

($\star$) for all $e_0, e_1 \in ES\backslash c_0$. $[e_0 \; R_0 \; e_1 \quad \Rightarrow \quad e_0 \in \downarrow(\uparrow e_1)]$

($\star$) follows by induction in i where $e_0 \; (\lessdot_0 \cup \gtrdot_0)^i \; e_1$. The basis step $i = 0$ is trivial. Assume $e_0 \; (\lessdot_0 \cup \gtrdot_0)^i \; e_2 \; (\lessdot_0 \cup \gtrdot_0) \; e_1$. From induction hypothesis there exists an e_3 such that $e_0 < e_3 > e_2$. In case $e_1 \lessdot_0 e_2$, we clearly have also $e_0 < e_3 > e_1$ and hence $e_0 \in \downarrow(\uparrow e_1)$. In case $e_2 \lessdot_0 e_1$, we have from above that $e_1 < g^k(e_2) < g^k(e_3)$. But we also have $e_0 < e_3 < g^k(e_3)$, and hence $e_0 \in \downarrow(\uparrow e_1)$.

Now to prove the second part of the lemma, suppose $e_0 \; R_0 \; e_1$. From Lemma 2, we get that $g^i(e_0) \; R_0 \; e_1$ for $0 \leq i \leq k$. Hence from ($\star$) above, $g^i(e_0) \in \downarrow(\uparrow e_1)$ for $0 \leq i \leq k$. On the other hand, from $e_0 \; co \; e_1$, we conclude that for $0 \leq i \leq k$, $g^i(e_0) \notin \downarrow e_1$. Now from Lemma 1 we get that there exists a j with $0 \leq j \leq k$ such that $g^j(e_0) \in \uparrow e_1$. Thus $e_1 \leq g^j(e_0) < g^{k+1}(e_0)$. □

We are finally ready to state how to extend any infinite regular conflict-free event structure ES into a regular conflict-free trace event structure. For a given ES let $k \in N$ and isomorphism $g : ES\backslash c_0 \rightarrow ES\backslash c_1$ be as fixed in the previous lemmas. Define the isomorphism $f = g^{k+1}$ and let F denote the equivalence on E induced by: If $e \in c_0$ then eFe' iff $e = e'$, and if $e \in E\backslash c_0$ then eFe' iff there exists $e_0 \in (E\backslash c_0)\backslash f(E\backslash c_0)$ and i, j such that $e = f^i(e_0)$ and $e' = f^j(e_0)$.

The required trace alphabet and labeling are now defined as follows:

[TR1] $\Sigma = \{ [e]_F \mid e \in E\}$,
[TR2] $[e_0]_F \; I \; [e_1]_F$ iff there exist $e'_0 \in [e_0]_F$ and $e'_1 \in [e_1]_F$ such that $e'_0 \; co \; e'_1$,
[TR3] for all $e \in E$. $\lambda(e) = [e]_F$.

We denote this Σ-labeled version of ES by LES. It is clear that Σ is a finite trace alphabet, containing one element for each element in $c_0 \cup ((E\backslash c_0)\backslash f(E\backslash c_0))$. We need to show that LES is indeed a trace event structure, and that it is regular. These two properties will be shown in the next two lemmas.

Lemma 4. *Let $ES = (E, \leq)$ be an infinite regular conflict-free event structure. Then LES as defined via $[TR1 - TR3]$ above is a conflict-free trace event structure.*

Proof. We need to show the following properties of LES:

- for all $e_0, e_1 \in E$. $e_0 \lessdot e_1 \quad \Rightarrow \quad \lambda(e_0) \; D \; \lambda(e_1)$,
- for all $e_0, e_1 \in E$. $\lambda(e_0) \; D \; \lambda(e_1) \quad \Rightarrow \quad e_0 \; (\leq \cup \geq) \; e_1$.

The second condition is clearly satisfied from the definition of (Σ, I) and λ.

The first condition is shown to hold by case analysis. Assume that $e_0 \lessdot e_1$.

Case 1. $e_0, e_1 \in c_0$

From $e_0, e_1 \in c_0$, it follows that $[e_0]_F = \{e_0\}$ and $[e_1]_F = \{e_1\}$, and from $e_0 \lessdot e_1$ it then follows that $[e_0]_F \; D \; [e_1]_F$.

Case 2. $e_0 \in c_0$ and $e_1 \in E\backslash c_0$

As in Case 1, we have $[e_0]_F = \{e_0\}$. From $e_0 \lessdot e_1$ and the construction of c_0 we get $e_1 \in (E\backslash c_0)\backslash f(E\backslash c_0)$ and thus $[e_1]_F = (\{f^i(e_1) \mid i \geq 0\}$. This implies that $e_0 \leq e'_1$ for all $e'_1 \in [e_1]_F$. Hence from the definition of I we conclude $[e_0]_F \; D \; [e_1]_F$, as required.

Case 3. $e_0, e_1 \in E\backslash c_0$

From $e_0 \lessdot e_1$ and Lemma 3 we get that $e_1 < f(e_0)$. From the definition of F it now follows that for all $e'_0 \in [e_0]_F$ and $e'_1 \in [e_1]_F$ either $e'_0 < e'_1$ or $e'_1 < e'_0$, and hence $[e_0]_F \; D \; [e_1]_F$, as required. □

Lemma 5. *Let $ES = (E, \leq)$ be an infinite regular conflict-free event structure. Then LES as defined by $[TR1 - TR3]$ above is a regular conflict-free trace event structure.*

Proof. Let us first construct inductively an (increasing) sequence of finite configurations $\{d_i\}$, $i \geq 0$, where $d_0 = c_0$ with c_0 being as in Lemma 2 and:

$$d_{i+1} = d_i \cup ((E\backslash d_i) \setminus (f(E\backslash d_i)).$$

The main goal of the following is to show that for all $c \in C_{LES}$ there exists a $c' \in C_{LES}$ such that $c' \subseteq d_2$ and $LES\backslash c$ is isomorphic to $LES\backslash c'$, from which regularity of LES clearly follows.

It turns out that the reachability relation R_0 (Definition 5) partitions $E\backslash d_0$ into equivalence classes with strong structural relationships to the configurations d_i, as formulated in the following two Claims. The first claim states that for any R_0-equivalence class, C_{R_0}, any finite configuration non-trivially intersects at most two consecutive sets of the form $C_{R_0} \cap d_i$.

Claim 1. Suppose $c \in C_{LES}$ and C_{R_0} is an R_0-equivalence class and $i \geq 2$. If $(C_{R_0} \cap E\backslash d_i) \cap c \neq \emptyset$ then $C_{R_0} \cap d_{i-1} \subseteq c$.

The arguments for Claim 1 go as follows. Assume that $e \in (C_{R_0} \cap E\backslash d_i) \cap c$ and that $e' \in C_{R_0} \cap d_{i-1}$. By definition, we have that $g^{-(k+1)}(e) \in E\backslash d_{i-1}$ and hence e' $(co \cup <)$ $g^{-(k+1)}(e)$. Suppose e' co $g^{-(k+1)}(e)$. From e' R_0 e we get from Lemma 3 that $e' < e$. In case $e' < g^{-(k+1)}(e)$ we get from Lemma 2 that $e' < e$. Now since c is downwards closed and $e \in c$ we conclude $e' \in c$.

The second claim states that for any finite configuration c, any set of the form $C_{R_0} \cap d_i$ with $i \geq 1$ splits $LES\backslash c$ into two independent sets of events.

Claim 2. Suppose $c \in C_{LES}$ and C_{R_0} is an R_0-equivalence class and $i \geq 1$. If $(C_{R_0} \cap E\backslash d_i) \cap c \neq \emptyset$ then $((E\backslash c) \cap C_{R_0})$ co $((E\backslash c) \setminus C_{R_0})$.

The arguments for Claim 2 go as follows. Let $e \in (C_{R_0} \cap E\backslash d_i) \cap c$ and $e_0 \in (E\backslash c) \cap C_{R_0}$, and $e_1 \in (E\backslash c)\backslash C_{R_0}$. We need to show that e_0 co e_1.

If $e_1 \in E\backslash d_0$ then clearly $e_0 \leq e_1$ (and $e_1 \leq e_0$) contradict the fact that $e_0 \in C_{R_0}$ and $e_1 \notin C_{R_0}$, and hence e_0 co e_1.

If $e_1 \in d_0$ then since $e_0 \in ES\backslash d_0$ we have $e_0 \not\leq e_1$. In order to show that also $e_1 \not\leq e_0$, consider $X = \{g^{-i}(e) \mid 0 \leq i \leq k\}$. Since $X \subseteq (E\backslash d_0) \cap c$ and $e_1 \in d_0 \cap E\backslash c$ we get that e_1 co X, and hence from Lemma 1 $X \not\subseteq \downarrow(\uparrow e_1)$. However since $e \in c$ and $e_0 \notin c$ we get that either $e < e_0$ or e co e_0, and in both cases (Lemma 2 and Lemma 3 respectively) we conclude that $e < g^{k+1}(e_0)$, i.e. $X \subseteq \downarrow g^{k+1}(e_0)$. So, it follows that $e_1 \not\leq g^{k+1}(e_0)$ and hence $e_1 \not\leq e_0$.

Now we are in a position to show the required regularity of LES, which will follow from

Claim 3 For all $c \in C_{LES}$ there exists a $c' \in C_{LES}$ such that $c' \subseteq d_2$ and $LES\backslash c$ is isomorphic to $LES\backslash c'$.

We argue for Claim 3 by induction in the size of c. What we shall show is that if c contains an element from $ES\backslash d_2$, then $ES\backslash c$ is isomorphic to $ES\backslash c'$, for some $c' \subset c$. By repeatedly applying this argument we will be able to conclude Claim 3.

Assume $e \in c \cap (E\backslash d_2)$. Define $c' = c \setminus f^{-1}((E\backslash c) \cap [e]_{R_0})$. It follows from Claim 1 that f^{-1} is totally defined on $(E\backslash c) \cap [e]_{R_0}$, and hence that c' is a well-defined proper sub-configuration of c. Furthermore, let us define $h : E\backslash c' \rightarrow E\backslash c$ as follows:

$h(x) = f(x)$ if $x \in (E\backslash c') \cap [e]_{R_0}$ and $h(x) = x$ otherwise.

Clearly, h is a well-defined function from $ES\backslash c'$ to $ES\backslash c$. We need to argue that h is indeed an isomorphism. From definition, h is label preserving and

bijective. So, we only have to show

$$\text{for all } \; e_0, e_1 \in ES\backslash c', e_0 \leq e_1 \text{ iff } h(e_0) \leq h(e_1).$$

Suppose $e_0 \leq e_1$. Clearly c' and $[e]_{R_0}$ satisfy the conditions of Claim 2. Hence assume that one of e_0, e_1 is in $[e]_{R_0}$ and the other not. Claim 2 would then imply that e_0 co e_1, contradicting $e_0 \leq e_1$. Thus both e_0 and e_1 are in $[e]_{R_0}$ or neither one is. This implies h is f on both the events or h is the identity on both events. In either case, $h(e_0) \leq h(e_1)$. Next suppose $h(e_0) \leq h(e_1)$. By a similar reasoning as above, we can conclude that $h(e_0) \leq h(e_1)$ implies $e_o \leq e_1$. □

Now we can finally conclude the first main theorem of this section:

Theorem 2. *Every regular conflict-free event structure ES is also a regular conflict-free trace event structure.*

Proof. If ES is finite the theorem is trivially true. If ES is infinite the theorem follows from the preceding two lemmas. □

The remaining part of this section is devoted to completing the proof of Theorem 1 by showing that every regular conflict-free trace event structure is isomorphic to the unfolding of a finite 1-safe Petri net.

It will be convenient to introduce the poset representation of traces. Fix through the rest of the section $M = (\Sigma, I)$, a trace alphabet with a, b ranging over Σ. The dependency relation of M is denoted as D and we let d range over D. A trace over this alphabet can also be represented as a Σ-labeled poset $Tr = (X, \leq, \lambda)$ where λ is required to satisfy :

- If $x \lessdot y$ then $\lambda(x)\ D\ \lambda(y)$ where D is the dependency relation induced by (Σ, I).
- If $\lambda(x)\ D\ \lambda(y)$ then $x \leq y$ or $y \leq x$.

Fix also an M-labeled regular conflict-free event structure $Tr = (E, \leq, \lambda)$. Then clearly, Tr can also be viewed as a trace over M which we shall do henceforth. We shall assume that E is an infinite set and let e, e' range over E. Our goal is to show that $(E, \leq)$ is isomorphic to the event structure unfolding of a finite (conflict-free) 1-safe Petri net.

Let $X, Y \subseteq E$. We shall say that X and Y are isomorphic iff the Σ-labeled posets $(X, \leq_X, \lambda_X)$ and $(Y, \leq_Y, \lambda_Y)$ are isomorphic where $\leq_X$ ($\leq_Y$) is $\leq$ restricted to X (Y) and λ_X (λ_Y) is λ restricted to X (Y). We shall say that $\pi \in E^\star$ is a linearization of X in case π is a linearization of the poset $(X, \leq_X)$. Next we define $L_X \subseteq \Sigma^\star$ as : $L_X = \{\lambda(\pi) \mid \pi \text{ is a linearization of } X\}$. (Here, by abuse of notation, the obvious extension of λ to $E^\star$ is also denoted as λ). Finally we set $L_{Tr} = \bigcup_{c \in C_{ES}} L_c$ with $L_\emptyset = \{\epsilon\}$. We will often write L_{Tr} as just L.

Let $\sigma \in L$. Then we associate with σ the configuration c_σ which has the property: $\sigma \in L_{c_\sigma}$. Clearly c_σ exists and it is easy to verify that c_σ is unique.

Lemma 6. *For any M-labeled regular conflict-free event structure Tr there exists an infinite sequence of configurations $c_0, c_1, c_2, \ldots$ of Tr such that the following conditions are satisfied.*

- *For every i and j if $i < j$ then $c_i \subset c_j$.*
- *For every i, j, $E\backslash c_i$ is isomorphic to $E\backslash c_j$.*
- *For every i, j, $c_{i+1}\backslash c_i$ is isomorphic to $c_{j+1}\backslash c_j$.*
- *$E = \bigcup_{i\geq 0} c_i$.*

The proof of this lemma is very similar to the proof of Lemma 2 and we omit it.

Let $d = (x, y) \in D$. We define Σ_d to be equal to $\{x\}$ if $x = y$ and equal to $\{x, y\}$ otherwise. For $\sigma \in \Sigma^\star$, $\sigma(d)$ will denote the Σ_d-projection of σ and it is the sequence obtained by erasing from σ the appearances of all letters except those of Σ_d. This notion is extended to L by defining $L(d)$ to be $L(d) = \{\sigma(d) \mid \sigma \in L\}$. The crucial observation now is that L is the synchronized product of the languages $\{L(d)\}_{d\in D}$. For more information on the theory of product languages see [HT].

For proving this product property of L, it will be convenient to extend λ to $\Sigma \times \{1, 2, \ldots\}$ as follows: $\widehat{\lambda}(e) = (\lambda(e), \mid \{e' \mid e' \leq e \text{ and } \lambda(e') = \lambda(e)\} \mid)$.

Lemma 7. *Let $\sigma \in \Sigma^\star$. Then $\sigma \in L$ iff $\sigma(d) \in L(d)$ for every d.*

Proof. Let $\sigma \in L$. Then $\sigma(d) \in L(d)$ for every d by definition. So assume $\sigma \in \Sigma^\star$ such that $\sigma(d) \in L(d)$ for every d. We need to show that $\sigma \in L$. We proceed by induction on the length of σ. If $\sigma = \epsilon$ then clearly $\sigma \in L$. So assume that $\sigma a \in \Sigma^\star$ and $\{\tau_d\}_{d\in D} \subseteq L$ are such that $\sigma a(d) = \tau_d(d)$ for every d. Clearly we can find for each d a prefix τ'_d of τ_d such that $\sigma(d) = \tau'_d(d)$. Since L is prefix-closed, each such $\tau'_d(d)$ is also in $L(d)$. Hence by the induction hypothesis, $\sigma \in L$.

Let i be the number of times the symbol a appears in σa. Then the symbol a appears in τ_{d_0} also i times where $d_0 = (a, a)$. Let c_0 be the configuration associated with $\tau_{d_0} \in L$. In other words, $c_0 = c_{\tau_{d_0}}$. Then there exists an event $e \in c_0$ such that $\widehat{\lambda}(e) = (a, i)$. Now let c be the configuration associated with σ. Then $e \notin c$ because the symbol a appears only $i - 1$ times in σ. If we show that $c' = c \cup \{e\}$ is also a configuration this would imply that $\sigma a \in L_{c'}$ and we are done.

So assume that $e' \lessdot e$ and $\lambda(e') = b$. Then $d_1 = (b, a) \in D$ and hence $\sigma a(d_1) = \tau_{d_1}(d_1)$. Let $\widehat{\lambda}(e') = (b, j)$. Then a appears i times in τ_{d_1} and hence $e \in c_1$ where c_1 is the configuration associated with τ_{d_1}. Consequently e' is in c_1 and thus the symbol b appears at least j times in τ_{d_1}. To be precise, it is j if $b \neq a$ and it is i if $b = a$ and in this case, $j = i - 1$. This shows that b also appears at least j times in σ which leads to $e' \in c$. We now have that $c \cup \{e\}$ is also a configuration. □

We now recall the infinite sequences of configurations claimed in Lemma 4 and fixing one such sequence we also fix for the rest of the section $\sigma_{fin} \in L_{c_0}$ and $\sigma_{inf} \in L_{c_1\backslash c_0}$.

Lemma 8.

- *For all $i \geq 0$, $\sigma_{fin}(\sigma_{inf})^i \in L_{c_i}$ where $\sigma^0_{inf} = \epsilon$.*

– *For all* $d \in D$, $L(d) = \bigcup_{i \geq 0} pref(\sigma_{fin}(d)(\sigma_{inf}(d))^i)$, *where* $pref(\tau)$ *is the set of prefixes of the sequence* τ.

Proof. The first part of the lemma follows easily from Lemma 4. To prove the second part, we first observe that $R.H.S \subseteq L.H.S$ follows easily from the fact that $\sigma_{fin}(\sigma_{inf})^i \in L_{c_i}$ for each $i \geq 0$. So assume that $\sigma \in L(d)$. Then there exists a configuration c and $\tau \in L_c$ such that $\tau(d) = \sigma$. By Lemma 4 there exists i such that $c \subseteq c_i$. Let $\tau' \in L_{c_i \setminus c}$. Then $\tau.\tau' \in L_{c_i}$. By a basic property of traces, $(\sigma_{fin}.(\sigma_{inf})^i)(d) = (\tau\tau')(d)$. The rest of the proof is routine. □

We now construct for each d a transition system TS_d such that the synchronized product of these transition systems yields the 1-safe net we are after. We define $TS_d = (S_d, \Sigma_d, \longrightarrow_d, s^d_{in})$ where

– $S_d = pref(\sigma_{fin}.\sigma_{inf}) \times \{\Sigma_d\}$,
– Σ_d is as before,
– $\longrightarrow_d \subseteq S_d \times \Sigma_d \times S_d$ is given by : $(\tau, \Sigma_d) \xrightarrow{a} (\tau', \Sigma_d)$ iff $\tau a = \tau'$ or $\tau a = \sigma_{fin}.\sigma_{inf}$ and $\tau' = \sigma_{fin}$,
– $s^d_{in} = (\epsilon, \Sigma_d)$.

For the sake of convenience, let us assume that $D = \{d_1, d_2, \ldots, d_n\}$ and replace subscripts of the form d_i by i. For instance TS_{d_i} will be written as TS_i etc. The synchronized product of $\{TS_i\}$ is denoted as $TS_1 \parallel TS_2 \ldots \parallel TS_n$ and its operational behavior is captured by the global transition system $TS = (S, \Sigma, \longrightarrow, s_{in})$ defined as follows:

– $S = S_1 \times S_2 \times \ldots \times S_n$,
– $\longrightarrow \subseteq S \times \Sigma \times S$ is given by $s \xrightarrow{a} s'$ iff for every d, $s[i] \xrightarrow{a}_d s'[i]$ if $a \in \Sigma_d$ (where $s[i]$ is the ith component of s etc.) and $s[i] = s'[i]$ otherwise,
– $s_{in} = (s^1_{in}, s^2_{in}, \ldots, s^n_{in})$.

We can associate the language $L_{TS} \subseteq \Sigma^\star$ with TS via :

– $\epsilon \in L_{TS}$ and $state(\epsilon) = s_{in}$,
– suppose $\sigma \in L_{TS}$, $state(\sigma) = s$ and $s \xrightarrow{a} s'$, then $\sigma a \in L_{TS}$ and $state(\sigma a) = s'$.

Let $\sigma \in \Sigma^\star$. Then it is easy to prove that $\sigma \in L_{TS}$ iff $\sigma(d) \in L(d)$ for each d. Thus $L_{TS} = L_{Tr}$ by Lemma 5 and all that remains to do is to associate the 1-safe Petri net with $TS_1 \parallel TS_2 \ldots \parallel TS_n$.

Let $\mathcal{N} = (B, T, F, M_{in})$ be the 1-safe Petri net given by:

– $B = \bigcup_{d \in D} S_d$,
– $T = \bigcup_{a \in \Sigma} T_a$ where T_a for each a is obtained via : Suppose $\{s_d\}_{a \in \Sigma_d}$ and $\{s'_d\}_{a \in \Sigma_d}$ are such that $s_d \xrightarrow{a} s'_d$ for each d satisfying $a \in \Sigma_d$, then $(\{s_d\}_{a \in \Sigma_d}, a, \{s'_d\}_{a \in \Sigma_d}) \in T_a$,
– $F = \bigcup_{t \in T} F_t$ where F_t for each t is given by :
Suppose $t = (\{s_d\}_{a \in \Sigma_d}, a, \{s'_d\}_{a \in \Sigma_d})$, then $F_t = (\{s_d\}_{a \in \Sigma_d}) \times \{t\}) \cup (\{t\} \times \{s'_d\}_{a \in \Sigma_d})$,
– $M_{in} = \{s^d_{in} \mid d \in D\}$.

It is easy to show that $\mathcal{N}$ is conflict-free. It is equally straightforward to show that the marking diagram of $\mathcal{N}$ is isomorphic to TS defined above. This then leads to the conclusion that the event structure unfolding of $\mathcal{N}$ is isomorphic to the underlying event structure of Tr thus establishing Theorem 1.

4 The Sequential Case

Here we wish to show that our conjecture goes through also for the sequential (co-free) case. This is very close to the classical case and hence is far easier to show. Nevertheless, it needs an argument because the folklore result involving regular trees and finite transitions systems deals with *ordered trees*. Before pinpointing this with an example, we first define co-free event structures.

Let $ES = (E, \leq, \#)$ be an event structure. Then ES is said to be *sequential* if for every $e, e' \in E$ it is the case that $e \leq e'$ or $e' \leq e$ or $e \# e'$. In other words the concurrency relation is empty. The notion of a 1-safe net being (behaviorally) sequential is defined in the expected manner: if at a reachable marking two elements are enabled then their F-neighborhoods are *not* disjoint. We wish to argue here that a sequential event structure is regular iff it is isomorphic to the event structure unfolding of a finite sequential 1-safe Petri net.

We note that the set of events enabled at a configuration do not have a pre-determined order and this is what makes the present case different from the classical case. For instance, $\{0,1\}^\star$ viewed as the full binary *ordered* tree has a $0-$*successor* and a $1-$*successor* at every node. But when viewed as a sequential event structure, at each configuration *two unordered* events are enabled.

To bring out the means for getting around this, fix a regular sequential event structure ES and let $C_{ES}/\ R_{ES} = \{[c_1], [c_2], \ldots, [c_n]\}$, where $[c]$ is the R_{ES}-equivalence class containing the configuration c. Fix $\prec$, a linear order over E, the set of events of ES.

Define now the finite alphabet Σ as : $\Sigma = \{[c_1], [c_2], \ldots, [c_n]\} \times \{1, 2, \ldots, d\}$ where d is the branching degree of ES (i.e. at every configuration at most d events are enabled).

We now associate the ordered (Σ-) tree (V, ρ) with ES as follows. V, the set of nodes of the tree, is the least subset of $\Sigma^\star$ given inductively by :

- $\epsilon \in V$ and $\mathit{config}(\epsilon) = \emptyset$.
- Suppose $\sigma \in V$ and $\mathit{config}(\sigma) = c$ and $a = ([c_i], j) \in \Sigma$. Then $\sigma a \in V$ iff there exists an event e enabled at c such that $\downarrow e \in [c_i]$ and $\mid \{e' \mid e' \in en(c)$ and $\downarrow e' \in [c_i]$ *and* $e' \prec e\} \mid = j - 1$. In this case, $\mathit{config}(\sigma a) = [c_i]$.

It is now routine to verify that V is a prefix-closed regular subset of $\Sigma^\star$ and thus (V, ρ) is a regular tree where $\rho \subseteq V \times V$ is the edge-relation given by $\sigma\ \rho\ \sigma'$ iff there exists $a \in \Sigma$ such that $\sigma' = \sigma a$. Consequently (folklore!), one can find a finite deterministic transition system $TS = (S, \Sigma, \longrightarrow, s_{in})$ such that the unfolding of TS is isomorphic to (V, ρ). It is now a routine exercise to extract from TS a finite conflict-free 1-safe Petri net whose event structure unfolding is isomorphic to ES.

It is, of course, trivial to establish that the event structure unfolding of a finite sequential 1-safe Petri net is a regular sequential event structure. Thus an event structure is regular and sequential iff it is isomorphic to the event structure unfolding of a finite sequential 1-safe Petri net.

5 Concluding Remarks

We have stated here a natural conjecture relating 1-safe Petri nets to prime event structures in the presence of a finiteness assumption. Our main result is that in the conflict-free case the conjecture indeed holds.

We have formulated our result for unlabeled event structures, but our result extends smoothly to structures labeled with an arbitrary (finite) alphabet. More interestingly, there exist a variety of models strongly related to 1-safe Petri nets, particularly in the setting of transition systems, as detailed in [WN]. We believe that suitable versions of our results hold uniformly across all these models. Moreover, we conjecture the same thing holds for our conjecture! Here we have concentrated on 1-safe Petri nets since it is one of the earliest and simplest concurrency models. Further, both the notions of a state and a transition are explicitly local and hence the conjecture is in a more demanding form.

The main surprise is that our conjecture seems so hard to prove. Even in the conflict-free case, the details involved in our proof are non-trivial. It is of course possible there is a simpler proof, but we fail to see it. It could also well be the case that the combinatorics underlying the proof of the main conjecture lie mainly in the conflict-free portion. One could then hope that a structural argument built on top of our proof of the conflict-free case will establish the full conjecture. At present however we see no evidence of this. Indeed, even the confusion-free case [NPW] does not seem easy to tackle. It is easy to exhibit an event structure which is confusion-free but in which an event is covered by an infinite number of other events. In other words, $e \lessdot e'$ for infinitely many e' and this at once destroys the argument we have used for settling the conflict-free case.

The best line of attack at present seems to be to prove that every regular event structure is also a regular trace event structure. It follows from [Thi2] that this is equivalent to the conjecture being true. A related labeling problem for event structures has been studied in [AC], but the label set is not a trace alphabet, and at this stage the exact relationship to our work is not clear.

References

[AC] R. Assous and C. Charetton: Nice Labeling of Event Structures, *Bulletin of the EATCS vol. 41*, (1990) 184-190. 350

[DR] V. Diekert and G. Rozenberg: *The Book of Traces*, World Scientific, Singapore (1995). 338

[E] J. Esparza: Model Checking Using Net Unfoldings, *Science of Computer Programming 23* (1996) 151-195. 335

[ERV] J. Esparza, S. Roemer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm, *Proc. of TACAS'96, Springer LNCS 1055* (1996) 87-106. 335

[HT] J. G. Henriksen and P. S. Thiagarajan: A Product Version of Dynamic Linear Time Temporal Logic, *Proceedings of CONCUR'97*, Springer LNCS 1243 (1997) 45-58. 347

[JN] M. Jurdzinski and M. Nielsen: Hereditary History Preserving Bisimilarity is Undecidable, *Proceedings of STACS 2000*, Springer Lecture Notes in Computer Science 1770 (2000) 358-369. 336

[Mc] K. L. McMillan: A Technique of a State Space Search Based on Unfolding, *Formal Methods in System Design 6(1)* (1995) 45-65. 335

[MT] P. Madhusudan and P. S. Thiagarajan : Branching-time Controllers for Discrete Event Systems. To appear in a special issue of *Theoretical Computer Science* containing full versions of selected papers from CONCUR'98. 336

[NPW] M. Nielsen, G. Plotkin and G. Winskel: Petri nets, Event structures and Domains I, *Theor. Comput. Sci. 13*, (1980) 86-108. 335, 337, 350

[NRT] M. Nielsen, G. Rozenberg and P. S. Thiagarajan : Transition Systems, Event Structures and Unfoldings, *Information and Computation, 118(2)*, (1995) 191-207. 335

[Thi1] P. S. Thiagarajan: Regular Trace Event Structures, *Technical Report, BRICS RS-96-32*, Computer Science Department, Aarhus University, Aarhus, Denmark (1996). 339

[Thi2] P. S. Thiagarajan: Regular Event Structures and Finite Petri Nets: A Conjecture, In: W. Brauer, H. Ehrig, J. Karhumaki, A. Salomma (Eds.) *Formal and Natural Computing : Essays Dedicated to Grzegorz Rozenberg*, Springer LNCS 2300 (2002) 244-253. 335, 350

[Tho] W. Thomas: Automata on infinite objects, In: J. van Leeuwen (ed.) *Handbook of Theoretical Computer Science, Volume B*, North-Holland, Amsterdam (1990) 130-191. 336, 337

[WN] G. Winskel and M. Nielsen: Models for Concurrency, *In: Handbook of Logic in Computer Science, Vol. IV*, Eds.: S. Abramsky, D. Gabbay and T. S. E. Maibaum, Oxford University Press, (1995). 335, 350

A Formal Service Specification for the Internet Open Trading Protocol*

Chun Ouyang, Lars Michael Kristensen**, and Jonathan Billington

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia, Mawson Lakes Campus, SA 5095, Australia
Chun.Ouyang@postgrads.unisa.edu.au
{Lars.Kristensen,Jonathan.Billington}@unisa.edu.au

Abstract. This paper presents our service specification for the Internet Open Trading Protocol (IOTP) developed using Coloured Petri Nets. To handle IOTP's complexity, we apply a protocol engineering methodology based on Open Systems Interconnection (OSI) principles consisting of five iterative steps: the definition of service primitives and parameters; the creation of an automaton specifying the local service language for each of the four trading roles of IOTP; the development of a CPN model synthesizing the local automata into a specification of the global service capturing the correlations between the service primitives at the distributed trading roles; the generation of the occurrence graph representing the global service language; and lastly a new step, language comparison to ensure the consistency between the specifications of the local service language and the global service language. The outcome is a proposed formal service specification for IOTP.

Keywords: System verification using nets, Case studies, Higher-level net models, Internet protocols, E-commerce.

1 Introduction

With the rapid growth of Internet-based electronic commerce (e-commerce), a multitude of communication protocols are being developed to build the required communication infrastructure. However, the design and analysis of these protocols is a difficult task that would benefit from the use of formal methods [24]. Formal methods such as Petri Nets [25] can be applied in a protocol engineering methodology [7,5]. Petri Nets have been successfully applied in the design and development of communication protocols to construct unambiguous specifications and allow specifications to be formally refined and analysed [6].

The *Internet Open Trading Protocol* (IOTP) [9,10] is an e-commerce protocol being developed by the Internet Engineering Task Force (IETF) [14]. IOTP is

* Partially supported by Australian Technology Network (ATN) Small Research Grant, 2001.
** Supported by the Danish Natural Science Research Council.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 352–373, 2002.
© Springer-Verlag Berlin Heidelberg 2002

a *shopping protocol* [8,17] focusing on business-to-consumer e-commerce. It is expected to become one of the central building blocks in next generation e-commerce, and a number of research groups and companies are working on the first trial implementations of IOTP [15,26,13] based on the informal *protocol* specification [9]. However, no complete implementation of IOTP exists, and the *service* provided to the users of IOTP has yet to be defined. The term *service* [1] refers to the capabilities provided by IOTP (and its underlying protocol layers) to its users. Developing a service specification represents an important step in developing protocols [7,5].

This paper applies Coloured Petri Nets (CPNs) [16] and Design/CPN [20] within a protocol engineering methodology [7,5] for developing a formal *service* specification for IOTP. Our previous work has been concerned with the abstract modelling and initial analysis of the *protocol* specification for IOTP [23], and a proposal for a complete and simplified *protocol* architecture [21]. Related work on formal service specification includes the OSI transport service [4], the Wireless Application Protocol (WAP) transaction service [12], and the Resource Reservation Protocol (RSVP) service [27].

The contribution of this paper is to give a formal specification of the trading service provided by IOTP, which we call the Internet Open Trading Service (IOTS). To manage the inherent complexity, we use a methodology that allows us to define the local interface behaviour before considering the correlations between the events at different interfaces. The methodology consists of five iterative steps for incrementally developing IOTS. Firstly, we define the events that can occur at each interface, known as service primitives. Next, Finite State Automata (FSA) [18] are used to define the local service languages for each of the four trading roles of IOTP. These represent all valid sequences of service primitives that can be observed at each *local* interface. Thirdly, a CPN is developed to synthesize the local automata into a specification of the *global* (distributed) service, which captures the correlations between the events at different interfaces. Fourthly, the occurrence graph (also known as the reachability graph or state space) of the CPN is generated, which represents all the possible sequences of service primitives at all four interfaces, which we call the global service language. Finally, language comparison is used to validate that the global service language is consistent with the local sequences of service primitives as specified in the local service languages.

The paper is organised as follows. Section 2 gives a brief introduction to the open trading service. The IOTS definition is presented in Section 3. It contains the definition of service primitives and parameters, and specifies the local service language for each trading role. Section 4 presents the IOTS CPN that defines the global service language. In Section 5, the approach to checking the consistency between the global service specification and the local service specification is presented. Finally, we summarize our contribution and discuss future work in Section 6.

2 Open Trading Service over the Internet

In this section, we informally describe the consumer-oriented open trading service over the Internet. The description is based on the mechanisms designed within the Internet Open Trading Protocol (IOTP). In IOTP terminology, the core service consists of five *trading transactions*. These transactions reflect the most common trading activities in the real world which are: *Purchase*, *Refund*, *Deposit*, *Withdrawal*, and *Value Exchange*. The *Purchase* is carried out for buying goods or services electronically. The *Refund* supports the refund of an earlier payment. The *Deposit* and *Withdrawal* transactions are used for the deposit or withdrawal of electronic cash by a customer at a financial institution. Finally, the *Value Exchange* transaction supports the conversion of an amount of electronic value in one currency to an amount of electronic value in another currency. A main design goal of IOTP is the encapsulation of payment protocols, e.g., *Secure Electronic Transaction* (SET) protocol[28] for credit card payments, and *Mondex Value Transfer Protocol* (VTP)[19] for Mondex electronic cash payments.

An online Purchase transaction occurs when one party wants to buy goods or services from another party over the Internet. Figure 1 shows a possible sequence of trading events in a Purchase transaction. Each column of the message sequence chart corresponds to a *trading role*. Four trading roles, namely *Consumer*, *Merchant*, *Payment Handler*, and *Delivery Handler* are defined in IOTP to identify the different roles in a trading transaction. For example, during a Purchase transaction, the *Consumer* receives and pays for goods or services; the *Merchant* supplies goods or services and receives payment via the *Payment Handler*; the *Payment Handler* (e.g., a bank) receives the money from the *Consumer* on behalf of the *Merchant*; and the *Delivery Handler* (e.g., a courier firm) delivers the goods to the *Consumer* on behalf of the *Merchant*. The Purchase transaction in Fig. 1 consists of a sequence of trading events labelled by event numbers in the order of the time when they occur. These trading events are further categorised into three subtransactions (or phases) which we call *trading exchanges* [21]. The three trading exchanges are *Offer*, *Payment*, and *Delivery*.

The first phase is the trading options *Offer* between Consumer and Merchant, regarding the payment method and payment protocol to be used. The Consumer decides to buy goods, and sends a Purchase Request (event 1) to the Merchant. Upon receiving the Purchase Request, the Merchant starts the transaction by offering the Consumer a list of Trading Options (event 2). This includes the available payment methods and associated payment protocols. The Consumer makes a selection among the options offered by the Merchant, and sends it back as a Trading Selection (event 3). The Merchant uses the Consumer selection to create and send back a Trading Result (event 4), which contains details of the goods to be purchased together with payment and delivery instructions.

The second phase is the *Payment* between Consumer and Payment Handler. After checking the Trading Result, the Consumer sends the Payment Handler a Payment Request (event 5) to invoke the payment. The Payment Handler checks the Payment Request, and if valid, carries out the payment via several Payment Data Exchanges (events 6 and 7). The number of payment data exchanges de-

pends on the payment protocol that is being used. The Payment Handler then sends a Payment Response (event 8) containing the payment result (e.g., receipt).

The third phase is the *Delivery* of goods. The Consumer checks the Payment Response and uses the payment receipt as an authorisation in a Delivery Request to the Delivery Handler (event 9). The Delivery Handler schedules the delivery and sends the Consumer a Delivery Response (event 10) containing details of the delivery, and possibly the goods (e.g., e-journal), if electronic. The Physical Goods Delivery followed by the Proof of Delivery is not part of IOTP v1.0.

Each of the trading exchanges may occur in different ways and as a result different variants of the transactions can be obtained. For an example, event 3 may or may not take place in an *Offer*, which results in the distinction between a *Brand Dependent Offer* and a *Brand Independent Offer* [9]. In another example, a delivery may be provided by a Delivery Handler or a Payment Handler. A *Delivery-Handler-provided Delivery* exchange occurs *after* the *Payment* exchange has completed, whereas a *Payment-Handler-provided Delivery* exchange occurs *during* the *Payment* exchange. Investigating all the transactions of IOTP shows that each of them can be expressed as combinations of *trading exchanges*. More specifically, by combining exchanges in different ways, we can obtain the trading transactions of all five types. A *Purchase* transaction consists of an *Offer*, a *Payment*, and optionally a *Delivery*. In a *Value Exchange* transaction, the *Offer* exchange is followed by two *Payment* exchanges that occur in sequence. The *Deposit*, *Withdrawal*, and *Refund* transactions start with an *Offer* exchange which is followed by a *Payment* exchange.

IOTP also supports the error recovery action taken by a Consumer role within a *Payment* exchange, and also the cancellation of an ongoing transaction by a trading role. We discuss both issues in the next section. Furthermore, IOTP defines three non-trading transactions, namely *Authentication*, *Status Inquiry*, and *Ping*. In this paper, we do not consider non-trading transactions. The reader is referred to [9,10] for details of the non-trading transactions.

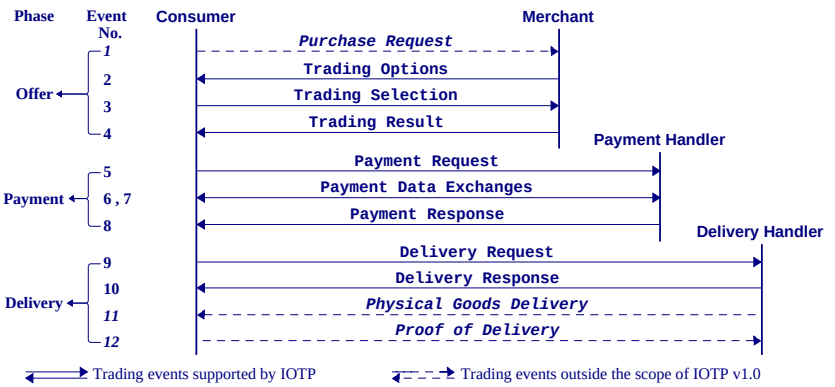


Fig. 1. A possible sequence of trading events in a Purchase transaction

3 Internet Open Trading Service Definition

Based on the general description of IOTP in Sect. 2, we now present a rigorous definition of IOTS. The definition has been developed using the conventions for OSI-service definitions [1]. IOTS defines the externally visible service provided by IOTP to trading roles.

3.1 IOTS Structure

The structure of IOTS is shown in Fig. 2. It comprises: IOTS users, i.e., trading role entities; the IOTS provider representing IOTP entities and all the communication layers below; four *service access points* (SAPs), each related to a trading role entity through which the service is provided by the IOTS provider to IOTS users; and submit and deliver *service primitives*. Service primitives describe the logical interactions between the service provider and service users. For example, a Consumer may request a service by *submitting* a service primitive, and the provider responds by *delivering* the corresponding service primitive to the appropriate peer trading role entity (e.g., the Merchant). A trading transaction (e.g., a Purchase) may involve *four* trading roles. This means that the IOTS is more complicated than, e.g., the OSI transport service [2], the WAP transaction service [12], and the RSVP service [27], each involving only *two* user entities.

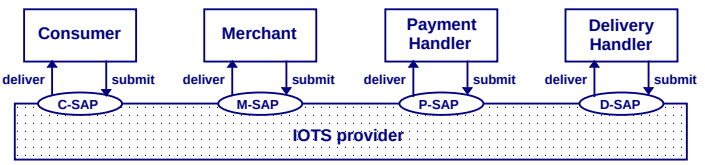


Fig. 2. The abstract IOTS structure

3.2 Service Primitives and Parameters

A trading transaction consists of a sequential combination of trading exchanges. We therefore define the service primitives relative to the trading service provided by each exchange rather than each transaction. To obtain a complete service definition, we also need to specify the parameters, which are associated with each service primitive and convey the required control information and user data. Table 1 lists the IOTS service primitives and their parameters. An abbreviated form (Abbr.) of each primitive is also given which we will use in later figures. These primitives are divided into four categories. The first three categories correspond to the three trading exchanges. In the last category, the two primitives represent the cancelling event that may occur during a transaction and constitute a *Cancellation* exchange. The conventions used to name a service primitive

Table 1. Service primitives and parameters

Exchange	Service	Primitive	Abbr.	Parameters
Offer	Trading offer	TradingOffer.req	TOFreq	(Transaction ID, transaction type, trading content, trading organisations)
		TradingOffer.ind	TOFind	(As above)
	Trading options	TradingOptions.req	TOPreq	(Transaction ID, transaction type, offer type, payment method options, trading organisations)
		TradingOptions.ind	TOPind	(As above)
		TradingOptions.res	TOPres	(Transaction ID, payment method selection)
		TradingOptions.cnf	TOPcnf	(As above)
	Trading result	TradingResult.req	TRreq	(Transaction ID, trading content, payment status, payment instructions, delivery status, delivery instructions, trading-role data)
		TradingResult.ind	TRind	(As above)
Payment	Payment invoke	PayInvoke.req	PIreq	(Transaction ID, trading-role data, payment instructions, payment method selection, payment method-specific data, delivery status, trading organisations)
		PayInvoke.ind	PIind	(As above)
	Payment data exchange	PayData.req	PDreq	(Transaction ID, payment method-specific data)
		PayData.ind	PDind	(As above)
		PayData.res	PDres	(As above)
		PayData.cnf	PDcnf	(As above)
	Payment result	PayResult.req	PRreq	(Transaction ID, trading-role data, process state, payment error code, payment receipt, payment note, payment method-specific data)
		PayResult.ind	PRind	(As above)
	Payment retry	PayRetry.req	RTreq	(Transaction ID, trading organisations, payment method selection, payment method-specific data)
		PayRetry.ind	RTind	(As above)
Delivery	Delivery	Delivery.req	Dreq	(Transaction ID, trading-role data, payment receipt, delivery instruction, order details, trading organisations)
		Delivery.ind	Dind	(As above)
		Delivery.res	Dres	(Transaction ID, delivery note, digital content data)
		Delivery.cnf	Dcnf	(As above)
Cancellation	Cancel	Cancel.req	Creq	(Transaction ID, cancel code)
		Cancel.ind	Cind	(As above)

conform to [1]. A detailed description of all IOTS primitives and parameters is given in [22]. Revisiting the scenario in Fig. 1, it can be seen that the two types of *trading offer* primitives (TradingOffer.req and TradingOffer.ind) correspond to event 1 which is not part of IOTP v1.0. We include it in IOTS since it seems appropriate to have Consumer initiated transactions.

3.3 Local Service Languages

We specify the IOTS that can be observed at each SAP by defining the sequences of service primitives that may occur at that SAP. We refer to this set of primitive sequences as the *local service language* of that SAP. State transition diagrams and primitive sequence tables have been used in other services for the specification of local service languages [2]. For the specification of the local IOTS languages we have created a finite state automaton (FSA) for each SAP. We have not used primitive sequence tables because their memory-less nature makes them inadequate for specifying the local IOTS languages.

Figure 3 shows the four FSAs that define the local service language at each SAP. To keep the presentation simple, we have omitted cancellation of transactions here. IDLE is the initial state in each of the automata, and acceptance (or halt) states are indicated by a thick solid border. States drawn with dashed round boxes and the primitives labelling dashed arcs are related to the error-recovery procedures during a Payment exchange. The definition of the local service language at the C-SAP (Fig. 3 (a)) comprises a set of C-SAP states and a set of primitive events that may occur at that C-SAP. Similarly, the primitive sequences defined at each of the other SAPs are shown in Fig. 3 (b)-(d).

The meaning of each C-SAP state is listed in Table 2. For brevity, we have omitted similar tables for the states of the other SAPs (see [22] for these tables). The primitives labelling the arcs are given in their abbreviated form as listed in Table 1. To keep the local service languages abstract, we allow any finite number of retries in the specification of the error-recovery procedure. Similarly, we allow any finite number of payment data exchanges which ensures that the local service languages are independent of the payment protocol being encapsulated. An example of an accepted sequence specified by the automaton in Fig. 3 (a) is (TOFreq, TOPind, TOPres, TRind, PIreq, PDind, PDres, PRind, Dreq, Dcnf). This sequence corresponds to the trading events at the Consumer side in the Purchase transaction scenario shown in Fig. 1.

We now briefly sketch how the cancellation of an ongoing transaction affects the local service languages. Cancellation is not included in Fig. 3 to avoid an unreadable complex diagram. For the results presented later, we have taken cancellation into account. Three cases need to be mentioned regarding the cancellation of an ongoing transaction. Firstly, a Consumer entity may cancel an ongoing transaction at any time by submitting a *Cancel.req*, and as result, a *Cancel.ind* is delivered to another non-Consumer trading role entity. Secondly, a non-Consumer entity can only cancel an ongoing transaction at *specific* points (using *Cancel.req*), and as a result, the Consumer is then informed of the cancellation (by means of *Cancel.ind*). Thirdly, the IOTS provider may abort an ongoing transaction at any time by issuing a *Cancel.ind* to both a Consumer entity and a non-Consumer entity. Therefore, primitives related to cancellation at the C-SAP can be represented by two arcs labelled by Creq and Cind leading from any C-SAP state (except IDLE and STOP) to the halt state STOP. Cancellation at non-C-SAPs is more complicated. Defining a *new* state STOP for M-SAP as the halt state of a cancelled transaction, it is specified by adding

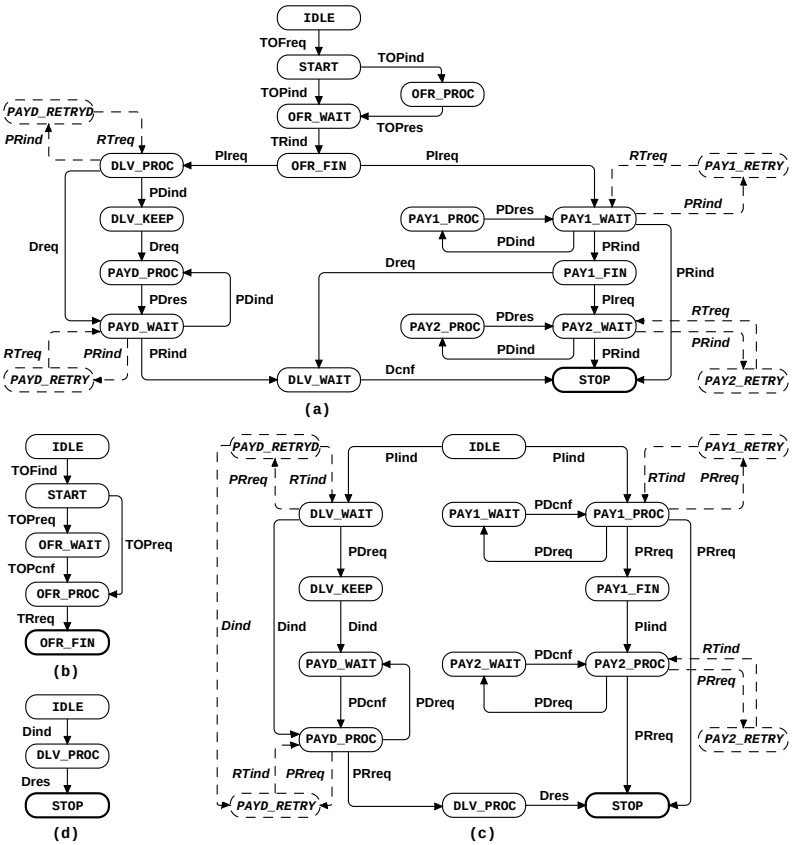


Fig. 3. FSAs for local service language at (a) C-SAP, (b) M-SAP, (c) P-SAP, and (d) D-SAP for transactions without cancellation

an arc labelled by Cind from any non-C-SAP state (except IDLE and STOP) to the halt state STOP, and an arc labelled by Creq from the *relevant states* in which a non-Consumer entity can cancel the ongoing transaction to the halt state STOP. These relevant states are START, OFR_PROC for M-SAP, PAY1_PROC, PAY2_PROC, DLV_WAIT, PAYD_PROC for P-SAP, and DLV_PROC for D-SAP. Additionally, there is also the cancellation between two trading exchanges, which means each non-C-SAP is either in the initial state or in a halt state. Due to space limitations, we will not discuss this issue further here.

4 Internet Open Trading Service CPN

The local service languages presented in the previous section specify only the primitive sequences at each of the SAPs, but do not capture the correlations

Table 2. Meaning of C-SAP states defined in the automaton in Fig. 3 (a)

State		Meaning
prefix	**generic**	**(in primitive sequence automaton)**
-	IDLE	The initial state reflecting there is no ongoing transaction.
-	START	A transaction is initiated.
OFR/PAY1/PAY2 /PAYD/DLV	WAIT	Waiting on the IOTS provider to deliver a service primitive, within the exchange indicated by the state prefix.
OFR/PAY1/PAY2 /PAYD	PROC	Waiting on the Consumer entity to submit a service primitive, within the exchange indicated by the state prefix.
OFR/PAY1	FIN	Successful completion of an exchange indicated by the state prefix.
DLV	PROC	Waiting on 1) the Consumer entity to request a delivery (Dreq); or 2) the IOTS provider to indicate a payment data exchange (PDind).
DLV	KEEP	To keep waiting on Consumer entity to request a delivery (Dreq).
PAY1/PAY2 PAYD	RETRY	Waiting on Consumer entity to retry a failed payment (RTreq), after Dreq has been submitted if a PAYD exchange is indicated.
PAYD	RETRYD	Waiting on Consumer entity to retry a failed payment (RTreq), only if Dreq has not yet been submitted in a PAYD exchange.
-	STOP	The final state indicating the end of a transaction.

Note: As a state prefix, OFR stands for Offer, PAY1 for 1st Payment, PAY2 for 2nd Payment, PAYD for Payment with Delivery, and DLV for Delivery.

between primitive events occurring at different SAPs. This section presents the global IOTS CPN. Its purpose is to specify the *global service language* which captures the correlations between service primitives at different SAPs.

4.1 Modelling Assumptions and Simplifications

We have made the following assumptions when specifying the IOTS.

A Single Transaction. In terms of primitive sequences, all five trading transactions are independent of each other. More specifically, any two trading transactions that are in progress do not interact with each other at the level of the protocol specification. Hence the primitive sequences of the two transactions do not affect each other. According to this, it suffices to develop an IOTS CPN model that considers each of the five transactions in isolation.

Significant Parameters. All primitives are taken into account, but only their *significant* parameters (see Table 1) are considered. The *significant* parameters refer to those that affect the sequencing of primitives and thereby the global service language. For example, the *Transaction ID* parameter can be ignored within each primitive, because of the previous restriction to a single transaction.

Communication. No communication takes place between non-Consumer entities. Communication is always between a Consumer entity and a non-Consumer entity. This is in accordance with [9].

Service Data Units (SDUs). Communication via the service provider is by means of SDUs. Since IOTP is currently designed to operate over HTTP[11], we assume within the service provider a reliable communication medium with no loss, no duplication, and no re-ordering. The only exception is when cancellation occurs. As defined in [9], the IOTS provider will stop all further processing of an ongoing transaction in case it is notified of a cancellation. We therefore assume that a *cancel* SDU can overtake any non-cancel SDU.

4.2 Overview of IOTS CPN

We specify the IOTS taking advantage of the hierarchical structuring mechanism of CPNs. Figure 4 depicts the top level of the IOTS CPN given by the IOTService page. This page has four places and five substitution transitions.

The three substitution transitions Offer Exchange, Payment Exchange, and Delivery Exchange are used to model the three trading exchanges. These constitute the basic transaction service. The two substitution transitions Error Recovery and Cancellation represent the error-recovery and the cancellation service, respectively. The place C_SAP is used to model the C-SAP where local sequences of primitives at the Consumer side can be viewed. Similarly, the three places M_SAP, P_SAP, and D_SAP represent the M-SAP, P-SAP, and D-SAP, respectively. These are called *SAP places*. Table 3 lists the definition of the colour sets used to model the states of trading-role SAPs. Colour set State (line 1) contains the states of all SAPs, and has four subsets: CState (line 2), MState (line 3), PState (line 4), and DState (line 5).

Each of these four subsets indicates the possible states of the corresponding trading-role SAP based on those defined in Fig. 3 and the description of cancellation given in Sect. 3.3. However, both C-SAP states and P-SAP states are simplified in the global IOTS CPN model compared to the ones specified in Fig. 3 (a) and (c), where only the state prefix PAY is used instead of the previous PAY1, PAY2, or PAYD defined in the payment-related SAP states. This can be done since the parameters of service primitives are explicitly presented in the global IOTS CPN model to guarantee the required synchronized end-to-end behaviour of trading role entities during a transaction. Some of these parameters convey the information indicating whether the 1st Payment, the 2nd Payment, or a Payment with Delivery is taking place.

4.3 Basic Transaction Service CPN

The basic transaction service consists of five trading transactions, where neither error-recovery nor cancellation can occur. It is modelled by the subpages of the three substitution transitions Offer Exchange, Payment Exchange, and Delivery Exchange.

Table 4 lists the definition of the color sets related to the SDUs and the significant parameters. From line 1 to line 6, six colour sets are defined for modelling the parameters (see Table 1) that are significant according to our assumptions in Sect. 4.1. These are: TransType (line 1) used for the *transaction type* parameter;

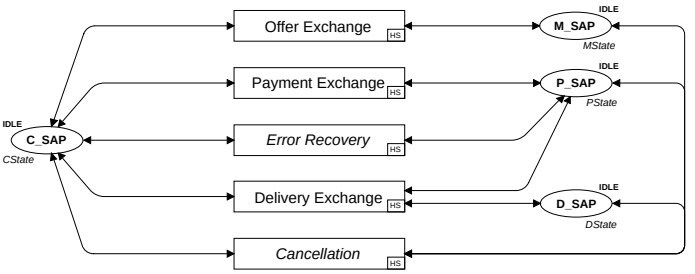


Fig. 4. Top level structure of IOTS CPN

Table 3. Colour sets and variables for modelling states of trading-role SAPs

```
1 color State = with IDLE | START | STOP | OFR_WAIT | OFR_PROC | OFR_FIN |
                    PAY_WAIT | PAY_PROC | PAY_FIN | PAY_RETRY | PAY_RETRYD |
                    DLV_WAIT | DLV_PROC | DLV_KEEP;
2 color CState = State;
3 color MState = subset State with [IDLE, START, OFR_WAIT, OFR_PROC, OFR_FIN, STOP];
4 color PState = subset State with [IDLE, PAY_WAIT, PAY_PROC, PAY_FIN, PAY_RETRY,
                                    PAY_RETRYD, DLV_WAIT, DLV_PROC, DLV_KEEP, STOP];
5 color DState = subset State with [IDLE, DLV_PROC, STOP];
6 var sc: CState;    var sm: MState;    var sp: PState;    var sd: DState;
```

OfferType (line 2) for the *offer type* parameter, indicating a Brand Dependent Offer (BDO) or a Brand Independent Offer (BIO); PayStatus (line 3) for the *payment status* parameter, indicating whether one or two Payment exchanges are involved; PayNumber (line 4) used to model the payment number information contained in the *payment instructions* parameter, indicating whether a Payment exchange is the first (PAY1) or the second (PAY2) in case there are two involved, otherwise this is not applicable (NA); DlvStatus (line 5) for the *delivery status* parameter, indicating whether the transaction has no delivery (ndlv), a Payment-Handler-provided Delivery (pdlv), or a Delivery-Handler-provided Delivery (ddlv); ProState (line 6) for the *process state* parameter, indicating whether a Payment exchange is successful (completedok) or has failed (failed). The colour sets PSxDS (line 7) and PNxDS (line 8) are defined as a product type, to model that the two corresponding parameters need to be carried in one SDU. The colour set IotSDU (line 9) models the complete set of SDUs used within IOTS, and the colour set IotSDUQ (line 10) is used to model the SDU queues.

To explain how we model the IOTS in detail, we focus on the Offer page, which is the subpage of the Offer Exchange substitution transition. Then we briefly describe the Payment page, which is the subpage of the Payment Exchange substitution transition. The Delivery page is similar and will not be discussed.

Figure 5 depicts the Offer page modelling the global sequences during an Offer exchange. It consists of 6 places and 8 transitions. The SAP places C_SAP

Table 4. Colour sets and variables for modelling SDUs

```
1  color TransType = with Purchase | Refund | Deposit | Withdrawal | ValueExchange;
2  color OfferType = with BDO | BIO;
3  color PayStatus = with one | two;
4  color PayNumber = with NA | PAY1 | PAY2;
5  color DlvStatus = with ndlv | pdlv | ddlv;
6  color ProState = with completedok | failed;
7  color PSxDS = product PayStatus * DlvStatus;
8  color PNxDS = product PayNumber * DlvStatus;
9  color IotSDU = union offrequest:TransType + options:OfferType + selection + offresponse:PSxDS +
                       payrequest:PNxDS + paydata + payresponse:ProState
                       dlvrequest + dlvresponse + payretry + cancel;
10 color IotSDUQ = list IotSDU;
11 var t:TransType; var f:OfferType; var p:PayStatus;  var n:PayNumber; var d:DlvStatus; var q:IotSDUQ;
```

and M_SAP are linked (via port-socket assignment) to the places with the same name in the IOTService page (see Fig. 5).

Service Primitive Transitions. The 8 transitions model the events of submitting and delivering service primitives, and are named accordingly. Each transition has an associated label (positioned above the transition) that specifies the abbreviated form of the corresponding service primitive as given in Table 1.

Service Provider Places. The places ConToMer and MerToCon represent communication between the Consumer and the Merchant. These places have the colour set IotSDUQ, indicating that each communication is modelled as delivering a SDU from a SDU queue.

Parameter Places. These are shown with dashed borders. The places C_Prmtrs and M_Prmtrs are used to store the significant parameters. For example, the place C_Prmtrs typed by the colour set PNxDS is where the *payment number* and *delivery status* parameters reside.

Arcs between SAP Places and Primitive Transitions. Each arc from a *SAP place* to a *primitive transition* has an inscription specifying the required state of the SAP for the primitive event to occur. The corresponding arc in the opposite direction has an inscription specifying the state of the SAP after the primitive event occurs. For example, the occurrence of the transition TradingOffer.req (top left of Fig. 5) changes the C-SAP state from IDLE to START.

Arcs between Provider Places and Primitive Transitions. Each arc specifies how the SDUs are delivered via the service provider, and each arc inscription has a variable q that represents the current list of SDUs within the service provider. This ensures that the SDUs are delivered in FIFO order.

Arcs between Parameter Places and Primitive Transitions. These are shown as dashed arcs. Each arc from a *parameter place* to a *primitive transition* has an inscription specifying the required value of the significant parameter(s)

for the primitive event to occur. The corresponding arc in the opposite direction has an inscription specifying the value of the significant parameter(s) being stored after the primitive event occurs. For an example, when a Purchase transaction is initiated, the primitive transition TradingOffer.ind (top right of Fig. 5) will occur in a binding with t=Purchase. As a result, a token with value (one,d) will be put into the parameter place M_Prmtrs, where d is bound to one of the three values of DlvStatus. In this case, there will be only one Payment exchange (p=one). In another example, after the transition TradingResult.ind (bottom left of Fig. 5) occurs, both payment number and delivery status information need to be stored into C_Prmtrs. The arc inscription indicates how payment status determines payment number information: if there is only one Payment exchange (p=one), the payment number will not be applicable (NA). Otherwise, the following Payment exchange will always be the first Payment exchange executed within the current transaction (PAY1).

Based on the above, we review the Purchase transaction scenario in Fig. 1, and explain how the Offer page captures the trading events in an Offer exchange. When the Purchase starts, two primitive transitions TradingOffer.req and TradingOffer.ind occur one after another, corresponding to event 1 in the scenario where a Purchase Request is sent from the Consumer to the Merchant. If a delivery is required and will be provided by a Delivery-Handler, a token with colour (one,ddlv) is then put into place M_Prmtrs, indicating the current Purchase transaction involves a Delivery-Handler-provided Delivery exchange. Occurrence of the next two transitions TradingOptions.req and TradingOptions.ind corresponds to event 2 in the scenario, where a list of Trading Options is offered by the Merchant to the Consumer and a Brand Dependent Offer is indicated in the SDU options(f=BDO). Next, TradingOptions.res and TradingOptions.cnf occur, modelling the Consumer's Trading Selection (event 3). Finally (event 4), the Merchant sends the Trading Result to the Consumer, represented by the occurrence of both TradingResult.req and TradingResult.ind transitions.

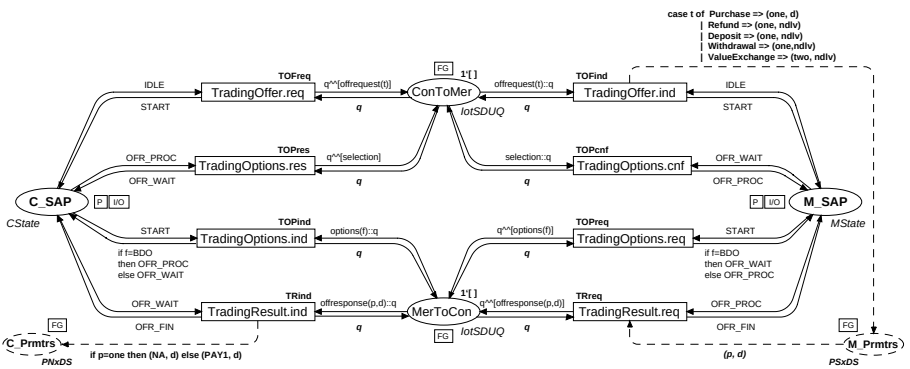


Fig. 5. Offer page in IOTS CPN

Figure 6 depicts the Payment page. It has a similar structure to the Offer page and is not described in detail except the inscription on the arc leading from transition PayResult.ind to place C_SAP, and the inscription on the arc leading from transition PayResult.req to place P_SAP. Both are used for the state assignment of C-SAP and P-SAP at the end of a Payment exchange. The state of a C-SAP or a P-SAP at the end of a Payment exchange is determined by the payment number (n) and the delivery status information (d). The parameter pair (n,d) may have five different values corresponding to the different scenarios of the transactions. For an example, the C-SAP will enter the state STOP if the transaction involves only one Payment exchange (n=NA) with no Delivery exchange (d=ndlv). As another example, the P-SAP will enter the state PAY_FIN when the first Payment exchange has been completed (n=PAY1) and no Delivery exchange follows (d=ndlv). It should also be mentioned that only successful Payment exchanges are considered in the Payment page. Unsuccessful Payment exchanges is part of error-recovery which we consider next.

4.4 Error-Recovery Service CPN

Figure 7 depicts the Error Recovery page modelling the global behaviour in the error recovery procedure due to a failed Payment exchange within an ongoing transaction. The error recovery procedure involves the C-SAP and the P-SAP. The occurrence of PayResult.req and PayResult.ind indicates that the current Payment exchange has failed, and the occurrence of the subsequent PayRetry.req and PayRetry.ind represents the Consumer retrying the payment with modified payment information according to the failure indication.

4.5 Cancellation Service CPN

The Cancellation page is used to model the global behaviour when cancelling an ongoing transaction. The cancelling event may occur at any time within an

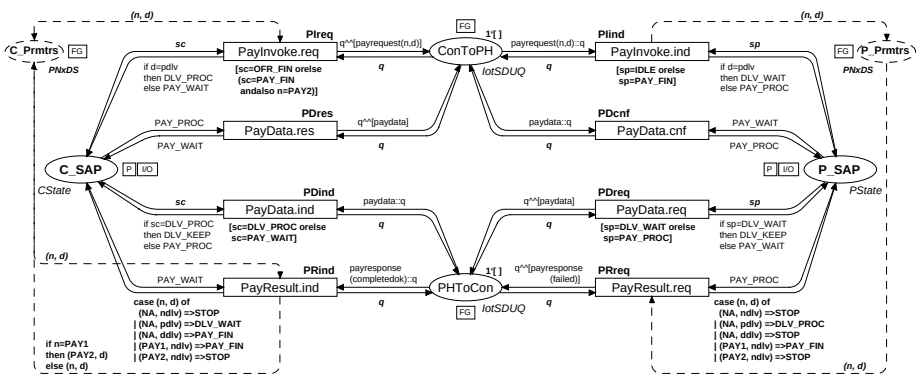


Fig. 6. Payment page in IOTS CPN

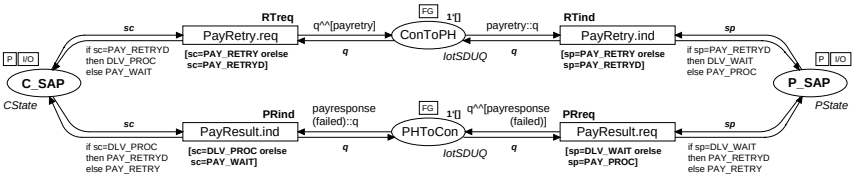


Fig. 7. Error Recovery page in IOTS CPN

ongoing transaction. Figure 8 depicts the part of the Cancellation page related to Offer exchange only. In Fig. 8, the four *cancel transitions* (drawn with solid borders) connected to solid arcs represent the cancellation initiated by trading roles, whereas the other two (drawn with dashed borders) connected to dashed arcs specify cancellation initiated by the IOTS provider. The enabling of each transition is restricted by the associated guard. For example, the Cancel.req transition (top left) can be enabled only when C-SAP is in one of the states specified in the guard. This ensures that the model specifies the allowed cancellations only. In addition, the delivery of a *cancel* SDU via the service provider is modelled as another single-element list [cancel] being put into and removed from a *service provider place*. The *cancel* SDUs are thereby maintained separately from non-cancel SDUs within the service provider. This is needed to properly specify that the delivery of a *cancel* SDU is independent of non-cancel SDUs, i.e., a *cancel* SDU may overtake a (list of) non-cancel SDU(s), in accordance with our modelling assumptions.

Cancellation within the other two trading exchanges is modelled in a similar way. However, we have applied more complicated modelling mechanisms with respect to those cancelling events that occur *between* the execution of two trading exchanges. We do not describe it further here because of space limitations.

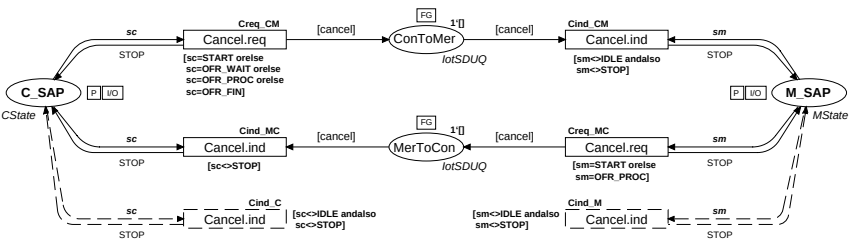


Fig. 8. Part of the Cancellation page in IOTS CPN

5 Internet Open Trading Service Analysis

The global service language is given by the occurrence graph of the IOTS CPN model, since the occurrence graph generated from the executable IOTS CPN model can be interpreted as a *finite state automaton* (FSA). The alphabet of the FSA corresponds to *binding elements* (transition modes) of the IOTS CPN model, and the acceptance (halt) states are defined by the dead (terminal) markings of the occurrence graph indicating the end of a transaction. A *binding element* of a CPN is a pair consisting of a transition and an assignment of data values to the variables of the transition.

The aim of our analysis is to ensure that our global service language as obtained from the IOTS CPN model in Sect. 4 is consistent with the local service languages as defined by the FSAs in Sect. 3. To check consistency we compare the local service languages at each SAP as specified in Sect. 3.3 to the global service language projected by hiding (making invisible) those primitive transitions that do not correspond to the SAP under consideration.

The FSM tool [3] is used to manipulate the service languages. This is achieved by converting the occurrence graph output from Design/CPN [20] into a text file that can be used as input to the FSM tool.

5.1 Global IOTS Language

The global IOTS language for the basic transaction service is obtained from the occurrence graph of the corresponding CPN model given in Sect. 4.3 when we do not consider error recovery and cancellation. The minimized FSA representing this global service language contains 42 states, 56 arcs, and 2 halt states, and is too complex to be shown graphically. However, it can be separated into four parts according to the four possible scenarios of the basic transaction service. The corresponding four FSAs obtained are shown in Figure 9 (a)-(d). The *union* of the *languages* represented by these four FSAs constitutes the complete global language for the basic transaction service.

In Fig. 9, the arc labels are specified using the abbreviated form of the service primitives (see Table 1). The node with number 0 (node 0) represents the *initial state*, and nodes drawn with double circles represent *halt states*. For brevity, FSA-(a) hereafter refers to the FSA shown in Fig. 9 (a), and so on.

FSA-(a) captures transactions with an Offer exchange followed by a single Payment exchange. This corresponds to the trading transactions *Deposit*, *Withdrawal*, *Refund*, and *Purchase without delivery*. The primitive sequences obtained from node 0 to node 9 correspond to an Offer Exchange, where the sequence through nodes (0,1,2,3,4,6,8,7,9) represents a Brand Dependent Offer exchange and the other two represent a Brand Independent Offer exchange. The subsequence (11,13,15,16,11) represents the payment data exchanges. It comprises the only cycle of FSA-(a).

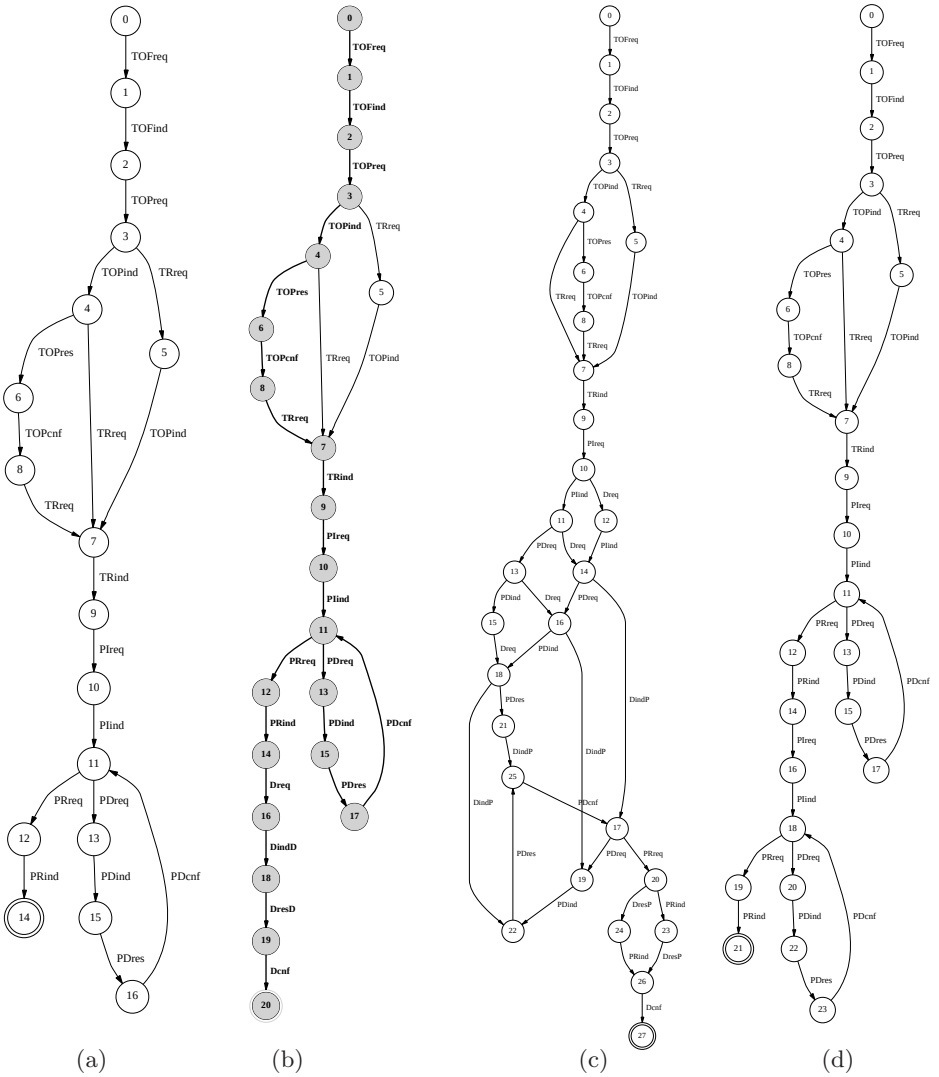


Fig. 9. Global FSAs of the basic transaction service with (a) a single Payment exchange only, (b) a Payment followed by a Delivery-Handler-provided Delivery exchange, (c) a Payment combined with a Payment-Handler-provided Delivery exchange, and (d) two Payment exchanges

FSA-(b) captures transactions with an Offer exchange and a Payment exchange followed by a Delivery exchange. This corresponds to the transaction *Purchase with payment before delivery*. In this case, the delivery is provided by the Delivery Handler and occurs after the payment has completed. The subsequence

(14,16,18,19,20) corresponds to a Delivery-Handler-provided Delivery exchange. As an example, the Purchase scenario given in Fig. 1 is represented by the sequence through the shaded nodes connected by thick arcs.

FSA-(c) captures the most complex scenario, where a Payment exchange is combined with a Delivery exchange that is provided by the Payment Handler. This corresponds to the transaction *Purchase with payment during delivery*.

FSA-(d) captures a transaction with two Payment exchanges. This corresponds to a *Value Exchange* transaction.

The global service language including error recovery and cancellation have been inspected in a similar way as above. The minimized FSA representing the global service language for transactions with error-recovery contains 58 states, 81 arcs, and 2 halt states. The minimized FSA for transactions with error recovery and cancellation contains 155 states, 570 arcs, and 4 halt states.

5.2 IOTS Language Comparison

One criteria in validating the IOTS specification is to ensure that the global service language consistently reflects the local service languages as specified for each trading-role SAP in Sect. 3.3 using *language comparison*. Two steps are required. Firstly, a *SAP projected-FSA* representing the local service language at the SAP is obtained from the global service language. It is generated by projecting the global FSA onto the set of primitives that can be viewed at that SAP only. Secondly, the SAP projected-FSA and the local SAP FSA (see fig. 3) are compared using the FSM tool. The purpose is to check the *language equivalence* between these two FSAs, i.e., whether or not the languages represented by the two FSAs are identical. These two steps are conducted for each of the four trading role SAPs.

Inconsistencies have been detected as a result of comparing the projected global service language against the local service language. The detected inconsistencies helped us to improve the IOTS CPN until all the inconsistencies were removed. With the refined CPN model, the global IOTS language matches the local service language at each trading-role SAP, and we are therefore highly confident in its correctness. Below we give a typical example of the inconsistencies detected by using language comparison and how the IOTS CPN was improved according to the detected inconsistencies.

Example: Missing Sequences in Transaction with Error-Recovery. Figure 10 (a) depicts a P-SAP projected-FSA obtained from the CPN model given in Sect. 4.4. The P-SAP FSA was obtained from the local service language specified in Fig. 3 (c). Comparison between these two FSAs reveals an inconsistency within the P-SAP projected FSA, where the Delivery.ind primitive (Dind) is missing from node 5 to node 10 (see Fig. 10 (a)). As a result, a valid sequence that has been defined within the local service language at the P-SAP is missing. Fig. 10 (b) depicts the valid sequence at the P-SAP that can be observed during a *Purchase with payment during delivery*. This primitive sequence

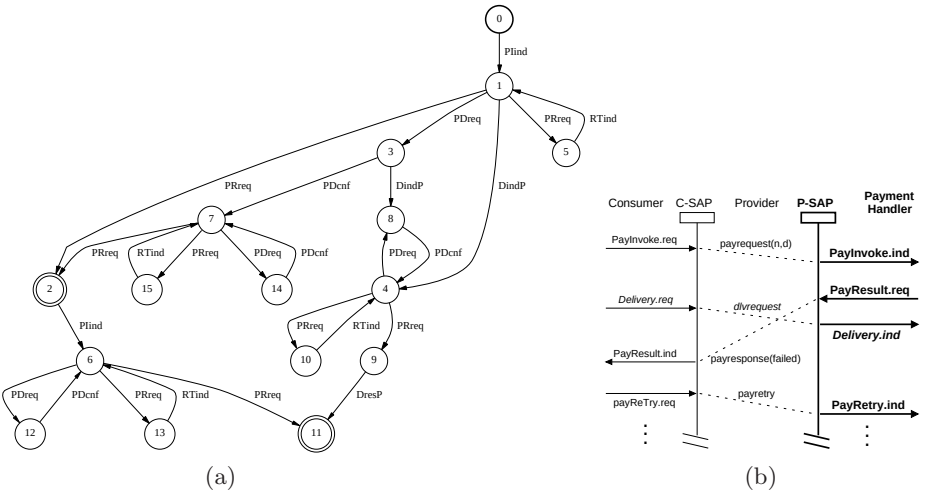


Fig. 10. Transaction with error-recovery service: (a) an inconsistent P-SAP projected FSA from the global FSA, and (b) a valid sequence at P-SAP not reflected by the FSA shown in (a)

is not presented in the FSA shown in Fig. 10 (a). The sequence starts with a PayInvoke.ind (node 0 to 1), followed by a PayResult.req (node 1 to 5), followed by a Delivery.ind (missing), and then a PayRetry.ind (node 5 to 1). The missing sequence in the P-SAP projected FSA hence reveals that there are missing sequences in the global FSA.

To remove this inconsistency, the IOTS CPN model is corrected. Transition Delivery.ind is added to the CPN of Fig. 7 as shown in Figure 11. In Fig. 11, the occurrence of the primitive transition Delivery.ind results in the change of the P-SAP state from PAY_RETRYD to PAY_RETRY. In this case, the Payment Handler is notified of the delivery request from the Consumer, and does not expect to receive it again. With this corrected CPN model, we obtain a consistent P-SAP projected FSA.

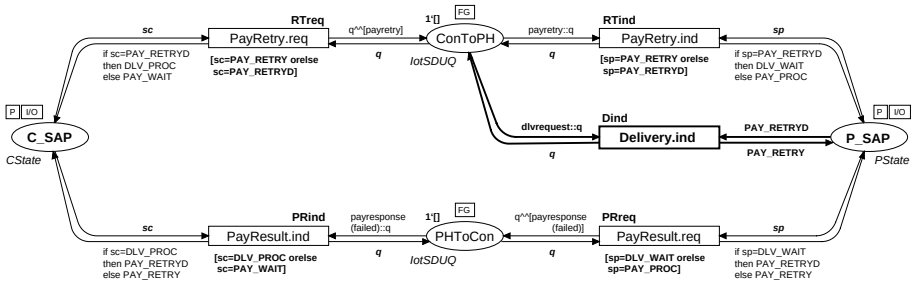


Fig. 11. Corrected Error Recovery page in IOTS CPN

6 Conclusions

We have developed a rigorous specification of the Internet Open Trading Service based on IOTP version 1.0 [9]. It comprises the definition of primitives and their parameters, and the set of valid local primitive sequences as viewed by each trading-role entity. Based on this definition, we have created the global IOTS CPN model to specify the global service language. The IOTS CPN model can be viewed as a formal service specification for IOTP.

The developed IOTS specification constitutes an essential step for IOTP verification. Developing the formal protocol specification for IOTP is another essential step. This has partially been done in [23,21], and its completion is planned as part of future work. IOTP verification will include comparing the sequences of primitives generated by IOTP with the IOTS service language.

We have also proposed an approach of applying language comparison to check for consistency between the global service language and the local service language. The purpose of conducting this language comparison is to make sure that the global service specification consistently reflects the local service specifications. This has led to the detection of inconsistencies which have been removed from the original CPN model.

Future work also involves investigating constructions for composing CPNs specifying local service languages in such a way that the local service languages are preserved, and the desired global sequences are obtained. This would eliminate the need for language comparison to ensure consistency between the local service languages and the projected global service languages.

Acknowledgements

The authors gratefully acknowledge the constructive comments of the reviewers on this paper.

References

1. *ITU-T Recommendation X.210, Information Technology - Open Systems Interaction - Basic Reference Model: Conventions For The Definition of OSI Services*, November 1993. 353, 356, 357
2. *ITU-T Recommendation X.214, Information Technology - Open Systems Interconnection - Transport Service Definition*, November 1995. 356, 358
3. AT&T. FSM Library. URL: `http://www.research.att.com/sw/tools/fsm`. 367
4. J. Billington. Abstract Specification of the ISO Transport Service Definition using Labelled Numerical Petri Nets. In *Protocol Specification, Testing, and Verification, III*, pages 173–185. Elsevier Science Publishers, 1983. 353
5. J. Billington. Formal Specification of Protocols: Protocol Engineering. In *Encyclopedia of Microcomputers*, pages 299–314. Marcel Dekker, New York, 1991. 352, 353
6. J. Billington, M. Diaz, and G. Rozenberg, editors. *Application of Petri Nets to Communication Networks: Advances in Petri Nets*, volume 1605 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1999. 352

7. J. Billington, M. C. Wilbur-Ham, and M. Y. Bearman. Automated Protocol Verification. In *Protocol Specification, Testing, and Verification, V*, pages 59–70. Elsevier Science Publishers, 1986. 352, 353
8. D. Birch. Shopping Protocols - Buying online is more than just paying. *Journal of Internet Banking and Commerce*, 3(1), January 1998. 353
9. D. Burdett. *Internet Open Trading Protocol - IOTP Version 1.0.* IETF Trade Working Group, April 2000. IETF RFC 2801 URL: http://www.ietf.org/rfc/rfc2801.txt. 352, 353, 355, 360, 361, 371
10. D. Burdett, D. Eastlake, and M. Goncalves. *Internet Open Trading Protocol.* McGraw-Hill, 2000. 352, 355
11. D. Eastlake and C. Smith. *Internet Open Trading Protocol (IOTP) HTTP Supplement.* IETF Trade Working Group, September 2000. IETF RFC 2935 URL: http://www.ietf.org/rfc/rfc2935.txt. 361
12. S. Gordon and J. Billington. Modelling the WAP Transaction Service using Coloured Petri Nets. In *Proceedings of MDA 1999*, LNCS 1748, pages 105–114. Springer-Verlag, 1999. 353, 356
13. InterPay I-OTP. URL: ftp://ftp.pothole.com/pub/ietf-trade/IETF-London/InterPay2001London.ppt%. 353
14. The Internet Engineering Task Force - IETF. URL: http://www.ietf.org/. 352
15. JOTP Open Trading Protocol Toolkit For Java. URL: http://www.livebiz.com/. 353
16. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 1-3.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 353
17. G. C. Kessler and N. T. Pritsky. Payment protocols: Cache on demand. *Information Security Magazine*, October 2000. Availabe via http://www.infosecuritymag.com/articles/october00/features2.shtml. 353
18. H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall International, 1998. 353
19. Mondex. URL: http://www.mondexusa.com/html/content/technolo/technolo.htm. 354
20. Design/CPN Online. URL: http://www.daimi.au.dk/designCPN/. 353, 367
21. C. Ouyang, L. M. Kristensen, and J. Billington. An Improved Architectural Specification of the Internet Open Trading Protocol. In *Proceedings of 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pages 119–137. DAIMI PB-554, Department of Computer Science, University of Aarhus, ISSN 0105-8517, 2001. 353, 354, 371
22. C. Ouyang, L. M. Kristensen, and J. Billington. A Proposed Internet Open Trading Service Definition. Technical report, Computer Systems Engineering Centre, University of South Australia, 2001. Draft version 2.0. 357, 358
23. C. Ouyang, L. M. Kristensen, and J. Billington. Towards Modelling and Analysis of the Internet Open Trading Protocol Transactions using Coloured Petri Nets. In *Proceedings of 11th Annual International Symposium of the International Council on System Engineering (INCOSE)*, pages CD–ROM 6.7.3, 2001. 353, 371
24. M. Papa, O. Bremer, J. Hale, and S. Shenoi. Formal Analysis of E-commerce Protocols. In *Proceedings of 5th. International Symposium on Autonomous Decentralized Systems*, pages 19–28. IEEE Computer Society, 2001. 352
25. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets: Advances in Petri Nets. Volume I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1998. 352

26. Hitachi SMILEs. URL: `http://www.hitachi.co.jp/Div/nfs/whats_new/smiles.html`. 353
27. M. E. Villapol and J. Billington. Generation of a Service Language for the Resource Reservation Protocol Using Formal Methods. In *Proceedings of 11th Annual International Symposium of the International Council on Systems Engineering (INCOSE)*, pages CD–ROM 9.1.4, 2001. 353, 356
28. Visa and MasterCard. *SET Secure Electronic Transaction Specification. Version 1.0. Vol 1-3*, May 1997. URL: `http://www.setco.org/set_specifications.html`. 354

Transition Refinement for Deriving a Distributed Minimum Weight Spanning Tree Algorithm

Sibylle Peuker

Software Verification Research Centre, The University of Queensland
Brisbane, Australia
sibylle@svrc.uq.edu.au

Abstract. The algorithm of Gallager, Humblet, and Spira computes the minimum spanning tree of a weighted graph in a distributed fashion. Though based on a simple mathematical idea, this algorithm is hard to understand because of the complex communication between the distributed agents.
There are several attempts to prove the correctness of this algorithm but most of the proofs are either incomplete or extremely long and technical and, therefore, difficult to comprehend.
This paper reports on a new model and a new correctness proof for the algorithm. Model and proof are given in a sequence of 12 property preserving refinement steps, starting with a simple non-distributed algorithm. All algorithms are modelled by Algebraic Petri nets. We justify 8 of the refinement steps using well-known refinement methods for distributed algorithms. The other four steps can not be justified using traditional refinement methods. Their correctness is proved by using the new concept of *transition refinement* which we introduced in [16].

1 Introduction

We introduced the concept of *transition refinement* in an earlier paper [16]. We now show the feasibility of transition refinement by applying it in a new correctness proof of a complex distributed algorithm. First, we motivate why we need yet another refinement concept.

The design of a distributed algorithm usually starts with a simple non-distributed algorithm that has some desired properties. That algorithm is stepwise refined to a fully distributed algorithm such that the desired properties are preserved in each step. In the refinement process, we are often faced with the problem that a synchronous action of several agents has to be distributed into asynchronous actions of the agents, e.g. by message passing.

A situation like this is shown in Fig. 1. There we have one agent, which changes its state from A to C, and another agent, which changes its state from B to D. On the left-hand side, this is done by a synchronous action of both agents; on the right-hand side the agents perform the state changes independently from each other. We now discuss why this situation is inconsistent with many notions of refinement.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 374–393, 2002.
© Springer-Verlag Berlin Heidelberg 2002

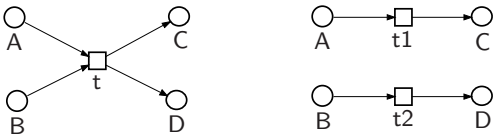


Fig. 1. A synchronous action of two agents and asynchronous actions of the agents

Many refinement notions are based on a sequential semantics of a system [1,5], i.e., a run of the system is given by a sequence of actions of the system. For the above situation this means that either t1 occurs before t2 or the other way round. A sequential semantics does not distinguish between independent events and events that occur non-deterministically in any order. The advantages of partial order semantics over sequential semantics for refinement were already discussed in [3]. In a partial order semantics, a run of the system is given by the causal order of occurring events. Independent events are not ordered, they occur concurrently. In particular, independence and non-determinism are distinguished.

Refinement notions usually do not allow the introduction of new non-determinism because the aim of the refinement process is to exclude non-determinism in order to obtain an implementable program. In our example, whether the replacement of t by $t1$ and $t2$ preserves properties of the system depends on the system which t belongs to. Assume that t is part of a system in which there is persistently a token on A but t is never enabled because there is never a token on B. After replacing t by $t1$ and $t2$, transition $t1$ may fire. If we assume that the token on A is reproduced after firing $t1$, then $t1$ may even fire infinitely often but $t2$ is never enabled. The replacement changes the behaviour of the system considerably and cannot be regarded as a correct refinement, not even based on a partial order semantics.

To deal with a situation like this, we proposed the concept of *transition refinement* [16]. Whether a replacement of a transition is a transition refinement depends on the replacement but also on the system that surrounds t. This concept is based on the causal order of actions in an algorithm. Because of their natural partial order semantics, Petri nets are well-suited for defining transition refinement. In [16], we introduce transition refinement for a simple class of Petri nets (elementary Petri nets). This Petri net class is not suitable for modelling complex distributed algorithms. Usually, Algebraic Petri nets are more adequate for this. Therefore, we develop the concept of transition refinement for Algebraic Petri nets in another paper [17]. We show that it is more powerful than transition refinement for elementary Petri nets. The refinement of a transition in an Algebraic Petri net corresponds to the simultaneous refinement of several transitions in an elementary Petri net.

In this paper, we show how to use transition refinement in the derivation of a complex distributed algorithm. The algorithm of Gallager, Humblet, and

Spira (GHS algorithm [7]) is a distributed algorithm for computing the minimum spanning tree of a weighted graph.

Because of its complexity, the algorithm became a yardstick for verification methods for distributed algorithms. The correctness of the algorithm was proved several times [4,9,12,22,25]. Two of the proofs [4,12] only prove the correctness of a deterministic version of the GHS algorithm. They do not deal with the non-determinism that is involved in the algorithm. The full version of the proof in [25] is very technical and about 200 pages long. [9] reports on a proof with a theorem prover. We discuss these proofs in more detail subsequent to our presentation.

Our proof consists of 12 property preserving refinement steps. The initial algorithm, a simple non-distributed algorithm, already computes the minimum spanning tree of the network. Informally, the properties preserved in each step are "The algorithm terminates and the final state shows the minimum spanning tree of the graph", even though the representation of the tree becomes more complex in the refinement process (from global data of the algorithm to local data distributed to the agents). We justify 8 of the refinement steps using well-known refinement methods for distributed algorithms. In these steps, we use a modified form of *refinement mappings* [1], data refinement [5], and for very simple steps the introduction of auxiliary variables [8]. The correctness of the other four steps cannot be proved using traditional refinement methods. We show that these steps are transition refinements.

In the refinement process, we use transition refinement to increase the *degree of distribution* of the algorithm, i.e., in each transition refinement step, we replace a synchronous action of agents by a small distributed algorithm that achieves the same goal as the synchronous action. In one step for instance, we replace an action where all agents of a group synchronously receive the same message by a distributed broadcast algorithm.

Using transition refinement, we are able to propose a complete new derivation of the algorithm which not only results in a well-structured proof but also gives new insight and assists in understanding this complex algorithm. Algebraic Petri nets [20] turned out to be a well-suited formalism for modelling this complex algorithm. The correctness of each refinement step is well illustrated by the change of the Petri net model. This increases the intuition for the formal proofs of the correctness of each step. Sometimes the change only consists of introducing a new variable that is appended to each token in the Petri net, and using the model, we can easily explain why this does not change the behaviour of the algorithm. In another case, the Petri net model is changed by the introduction of a new transition that is only connected to one place. This illustrates that the change is local and affects only a small part of the algorithm.

Due to space limitations, we cannot explain all 12 refinement steps in full detail. We explain our initial model and the first refinement step to illustrate the use of Algebraic Petri nets as a modelling formalism. Afterwards, we discuss how transition refinement is used in the derivation of the algorithm. The paper contains Petri net models of selected algorithms of the refinement hierarchy to illustrate how the algorithm evolves and to get an idea of its complexity.

The technical report that extends this paper [18] contains an appendix that includes the Petri net models of all algorithms of the refinement hierarchy. The full proof is given in [19] in 45 pages, including 31 illustrations of Algebraic Petri nets and their behaviour together with detailed informal descriptions.

The paper is organised as follows: In Section 2, we state the problem that is solved by the GHS algorithm. In Section 3, we give a simple, non-distributed solution, which is the starting point of our refinement hierarchy. In Section 4, we explain the first refinement step (which is not a transition refinement). In Section 5, we informally explain the concept of transition refinement and how it can be used in the verification of distributed algorithms. In Section 6, we concentrate on the transition refinement steps in the derivation of the GHS algorithm. In Section 7, we give a brief overview of existing proofs for the correctness of the algorithm. Finally, we compare our derivation with these proofs.

2 The Problem

We consider a uniquely weighted connected communication network which consists of a finite set of agents, a set of bidirectional communication channels between agents, and a weight function that assigns a weight (a positive real number) to each channel such that different channels have different weights. Two agents that are connected via a channel are called *neighbours*. Neighbours can send messages to each other. We assume that each agent knows its neighbours and the weights of all channels to its neighbours.

Mathematically, such a communication network is an undirected, connected, weighted graph (A, N, w) which consists of a set of vertices A, a set of edges $N \subseteq A \times A$ which is a symmetric relation, and a weight function $w : N \longrightarrow \mathbb{R}^+$. We represent a channel between agents x and y of A by the two pairs (x, y) and (y, x), one pair for each direction. We use *agent*, *channel*, and *network* synonymously for *vertex*, *edge*, and *graph*.

A *spanning tree* is an undirected, acyclic, connected subgraph of the graph which underlies the network. A *minimum spanning tree* is a spanning tree such that the sum of weights of edges of the tree is minimal, compared to all other spanning trees. A simple result from graph theory is: Each undirected, connected, uniquely weighted graph has a unique spanning tree [7]. The GHS algorithm computes the minimum spanning tree of the given graph in a distributed fashion.

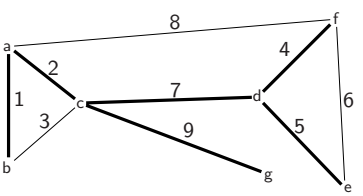


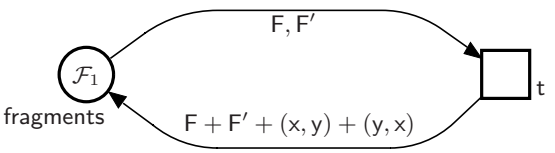
Fig. 2. A weighted communication network with minimum spanning tree (bold)

Here, *distributed* means that each agent executes a protocol based on its local knowledge and communicates by asynchronous message passing with its neighbours. In the end, each agent knows which of its adjacent channels belong to the minimum spanning and which do not.

3 A Simple (Non-distributed) Solution

We consider an undirected, connected, uniquely weighted graph as described in the previous section. A *fragment* is a non-empty subtree of the (uniquely determined) minimum spanning tree of the graph. A fragment that consists of only one vertex and no edge is called a *trivial fragment*. The *minimal outgoing edge* of a fragment is the edge of the graph with the least weight that connects an agent of the fragment with an agent outside the fragment (if there is any). The GHS algorithm is based on the following fact: The minimal outgoing edge of a fragment belongs to the minimum spanning tree of the graph. An easy proof of this fact is given in [7].

From this fact, we can derive several simple non-distributed algorithms for computing the minimum spanning tree of a graph. We use the following algorithm for starting our refinement hierarchy: we start with the set of trivial fragments. If two fragments are connected by an edge that is the minimal outgoing edge for at least one of the fragments then we combine the two fragments to form a new fragment. The new fragment consists of all vertices and edges of the old fragments plus the connecting edge of the old fragments. We repeat this step until there is only one fragment left (which has no outgoing edge). This fragment is the minimum spanning tree.



sorts		constants
F, F′ :	$\mathbb{F}$ragment	$\mathcal{F}_1 = A \times \{\emptyset\}$
x, y :	A	
fragments :	$\mathbb{F}$ragment	

transition guard

t : $x \in F \wedge y \in F' \wedge (y, x) = \mathrm{mo}(F')$

Fig. 3. The initial model Σ_1

A Petri net model of this algorithm is given in Fig. 3. We assume that the reader is familiar with Algebraic Petri nets [20] and their partial order semantics given by concurrent runs [13]. In this model (and in all subsequent models) we use the following notation:

We denote the minimum spanning tree of the graph (A, N, w) by $\mathrm{mst}(A, N, w)$. By the sort $\mathbb{F}\mathrm{ragment} = 2^A \times 2^N$, we denote the set of all pairs that consist of a set of agents and a set of edges of the graph. We defined a fragment as a subtree of $\mathrm{mst}(A, N, w)$. Hence, each fragment is of the sort $\mathbb{F}\mathrm{ragment}$. The sort $\mathbb{F}\mathrm{ragment}$, however, also includes substructures that are not fragments. Let $F = (V, E)$ be an element of $\mathbb{F}\mathrm{ragment}$. For an agent $x \in V$ and an edge $(x, y) \in E$, we also write $x \in F$ and $(x, y) \in F$. If F has an outgoing edge, we denote the minimum outgoing edge of F by $\mathrm{mo}(F)$. If F includes all agents of the network ($V = A$) then $\mathrm{mo}(F)$ is not defined.

We define the addition of two fragments $F = (V, E)$ and $F' = (V', E')$ as the union of their agents and edges: $F + F' = (V \cup V', E \cup E')$. We use $F + (x, y)$ as an abbreviation of $F + (\emptyset, \{(x, y)\})$, the addition of a fragment with an edge.

The initial marking of the model Σ_1 is the set $\mathcal{F}_1$ of all trivial fragments. Transition t is enabled if two fragments F and F' from the place fragments satisfy the transition guard of t. In the transition guard, we formalise the requirement that both fragments have a connecting edge which is the minimal outgoing edge of at least one of the fragments. By firing transition t, the fragments F and F' unite to form a new fragment $F + F' + (x, y) + (y, x)$.

In this algorithm, several new fragments in different parts of the network may be constructed concurrently to each other. Fig. 4 shows concurrent runs of Σ_1 for the example network of Fig. 2. Causally independent events are not ordered. In run ρ_1 (left-hand side) the trivial fragments of a and b merge into a new fragment. This happens concurrently to the events that merge the trivial fragments c and g and the trivial fragments f and d, respectively, into new fragments.

The correctness of this algorithm can be specified by two properties: (1) If there is only one fragment, then this fragment is the minimum spanning tree.

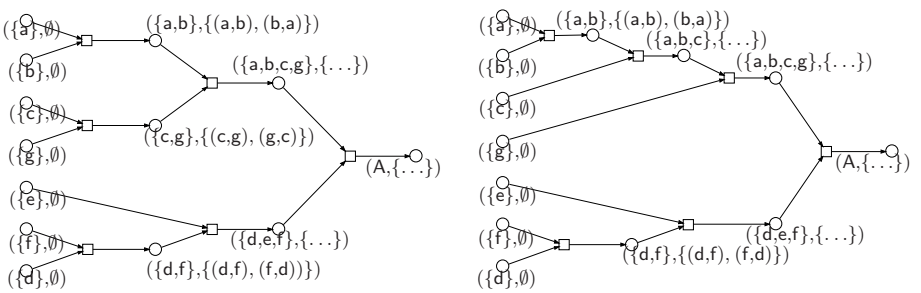


Fig. 4. Two concurrent runs ρ_1 and ρ_2 of Σ_1

(2) Eventually there will be only one fragment. The correctness is easy to prove using the mathematical facts given before.

4 The First Refinement Step

In the first refinement step, we introduce a new variable for each fragment. The value of this variable is always a natural number and we call it the *level* of the fragment. All trivial fragments have the level 0. The level restricts the construction of new fragments in the following way.

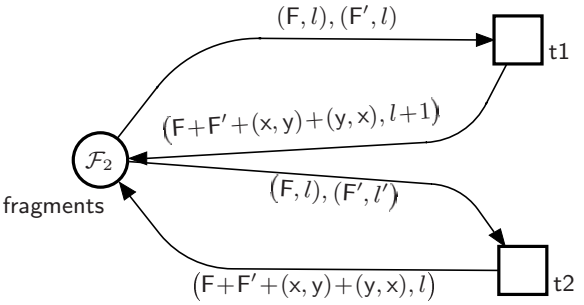
(i) **Merging:** Both fragments have the same level l and the same minimal outgoing edge (a, b) and (b, a), respectively. The new fragment has the level $l + 1$.
(ii) **Joining:** The fragments have different levels and the minimal outgoing edge of the fragment with the lower level connects both fragments. The level of the new fragment is the level of the fragment with the higher level.

The importance of the levels in the GHS algorithm only becomes visible in a later refinement step, when an agent (only using local knowledge and messages from its neighbours) has to decide if an adjacent edge is going out of its fragment or not. The level then becomes a measure for the timeliness of the information obtained from a neighbour.

A Petri net model of this second algorithm is given in Fig. 5. Transition t1 corresponds to the merging of two fragments, transition t2 corresponds to a lower level fragment joining a higher level fragment. This system has fewer concurrent runs than Σ_1 but each run of Σ_2 can be matched to a run of Σ_1 by omitting the levels.

The run ρ_2 of Σ_1 in Fig. 4 becomes a run of Σ_2 if we extend the labels to include appropriate levels: all trivial fragments are of level 0, all intermediate fragments with 2,3, or 4 agents are of level 1, and the final fragment (the minimum spanning tree) is of level 2. The run ρ_1 of Σ_1 has no corresponding run in Σ_2. The restriction imposed by the levels does not allow the combination of the trivial fragments of c and g because they have the same level and the edge with weight 9 is the minimal outgoing edge of g but not of c.

This refinement step is not a transition refinement. With a usual simulation technique [5], we can show that Σ_2 is a simple data refinement of Σ_1. To ensure that the liveness property ("eventually there is only one fragment") is preserved, we have to show, that the restriction imposed by the levels does not lead to a deadlock before the minimum spanning tree is computed. This is done by mathematical reasoning about all fragments with minimal level in a state of deadlock. Their minimal outgoing edges cannot form a circle. Hence, there must be one fragment with minimal level and a minimal outgoing edge leading into a fragment with higher level (then t2 may fire) or two fragments with minimal level have the same minimal outgoing edge (then t1 may fire).



sorts

l, l' : $\mathbb{N}$

fragments : $\mathbb{F}$ragment $\times\mathbb{N}$

All other sorts as in Σ_1.

constant

$\mathcal{F}_2 = A \times \{\emptyset\} \times \{0\}$

transition guards

t1 : $g_1 \wedge (x, y) = \text{mo}(F)$

t2 : $g_1 \wedge l' < l$

$g_1 \equiv \big(x \in F \wedge y \in F' \wedge (y, x) = \text{mo}(F')\big)$ is the transition guard of Σ_1.

Fig. 5. Σ_2

5 Transition Refinement and Distributed Algorithms

We will now explain transition refinement for Petri nets at a rather abstract level. The concept of transition refinement, and in particular, transition refinement for Algebraic Petri nets was formally introduced earlier [16,17].

We consider a Petri net N with an initial marking M_0. We call the pair $\Sigma = (N, M_0)$ a system. We present the behaviour of a system by *concurrent runs* (or short: runs) as illustrated in the previous section (for a formal definition see e.g. [13] or [17]).

Let t be a transition of the system. A *replacement* for t in Σ is a Petri net R that contains the places of the pre- and postset of t and that has a similar input/output behaviour to t, i.e. if R is initially marked with the preset of t then each run of R will terminate and in the end R will be marked with the postset of t (as in Fig. 1).

We can now straightforwardly replace t by R in Σ by deleting t and connecting R with Σ via the places from the pre- and postset of t. We obtain the *extended system* $\Sigma[t \to R]$. From the fact that t and R behave similarly in isolation, we cannot conclude that Σ and $\Sigma[t \to R]$ behave similarly, i.e. that the

replacement preserves the important properties of Σ. We will call a replacement that does preserve properties a *transition refinement*.

Before we can explain when a replacement is a refinement, we need another notion: Let ρ be a concurrent run of Σ. Similarly to the system itself, we replace each occurrence of the transition t in ρ by a concurrent run of R. We call the result a $[t \to R]$-*expansion* of ρ. A replacement R of a transition t in Σ is a transition refinement if each $[t \to R]$-expansion of a run of Σ is a run of $\Sigma[t \to R]$ and, conversely, if each run of $\Sigma[t \to R]$ may be obtained by expanding a run of Σ as described.

In an earlier paper [16], we give an equivalent formulation of transition refinement that involves the notion of *cuts*. A cut is a distributed global state. Formally, it is a maximal set of independent (i.e. unordered) conditions in a concurrent run of the system. Informally, a replacement of a transition is a transition refinement if in each run of the extended system all events that belong to the replacement can be partitioned into equivalence classes such that for each equivalence class, we can find two cuts such that the events between the cuts are exactly the events of this class. This means, in each run of the extended system, we can "cut out" all runs of the replacement by cuts.

A transition refinement preserves many important properties of the system. All liveness properties of the form "eventually something good will happen" are preserved. Safety properties of the form "never something bad will happen" are generally not preserved but the safety properties of the new system may be composed of the safety properties of the old system and the safety properties of the replacement. Alpern and Schneider [2] show that the correctness of any distributed algorithm can be stated in terms of liveness and safety properties[1]. For a more elaborate and formal discussion of this topic see [19,16]. For the purposes of this paper it is sufficient to know that if the old system always terminates and delivers a certain result then the new system also always terminates and delivers the same result.

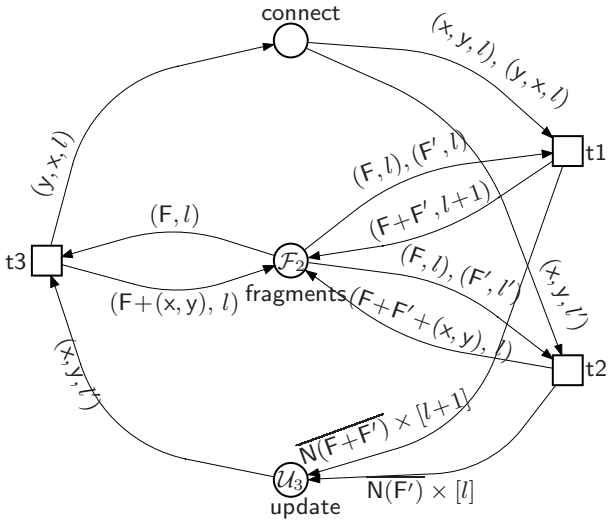
6 The Derivation of the GHS Algorithm

Communication between Fragments

The derivation of the algorithm consists of 12 property preserving refinement steps. The fifth refinement step is the first transition refinement step. Steps 2, 3 and 4 are easy steps, similar to the first step. Therefore, we only give a brief overview of these three steps in order to explain the fifth step.

In Algorithm 3 (Fig. 6), we introduce the communication between fragments via message passing. Each fragment determines its minimal outgoing edge. Then it sends a connect-message via this edge to inform the fragment at the other end of the edge that both fragments should unite.

[1] Considering distributed algorithms, we use the notions of liveness and safety in the sense of Alpern and Schneider as described above. These notions are not to be confused with the notions of liveness and safety usually used in Petri net theory.



sorts	constants
connect: $N \times \mathbb{N}$	$\mathcal{U}_3 = N \times \{0\}$
update: $N \times \mathbb{N}$	

The other sorts, and $\mathcal{F}_2$ as in Σ_2.

transition guards

t1 : $x \in F \wedge y \in F'$
t2 : $x \in F \wedge y \in F' \wedge l' < l$
t3 : $(x, y) \notin F \wedge (x, y) = \mathrm{mo}(F) \wedge l \leq l'$

Fig. 6. Σ_3; Communication between fragments

To keep combinations of fragments restricted to the cases *merge* and *join*, we need to ensure that the level of the fragment sending a connect- message is equal to or less than the level of the fragment that receives the message. Therefore, each newly constructed fragment sends update-messages to all neighbouring agents to inform them of its new level. A connect-message may only be sent after an update-message with at least an equal level was received (transition t3).

Two fragments merge if they have sent connect-messages to each other (transition t1). By the creation of connect-messages, this implies already that both fragments have the same level. A fragment joins a fragment with a higher level after it has sent a connect-message to the higher level fragment (transition t2) .

In Algorithm 4, we introduce a new variable, the name of the fragment. The name of a trivial fragment equals the name of its only agent. If two fragments

connect to form a new fragment, the new fragment gets the name of one of the agents adjacent to the connecting edge. We assume that there is a lexicographic order between agent names and assign the name of the larger agent to the fragment. The fragment name is an identifier of the fragment. This is important to distribute the algorithm further. In the final algorithm, each agent has to find its minimal adjacent outgoing edge. This is compared to the minimal outgoing edges of all other agents of the fragment. To determine if an adjacent edge is going out of the fragment or not, an agent uses its fragment name and its level. However, in Algorithm 4, the fragment name can be seen as an auxiliary variable that does not influence the behaviour of the algorithm.

In Algorithm 5, we distribute the data structure. Until now, we considered each fragment as a processor which stored data including name and level. From now on, we consider each agent of the network as a processor which locally stores data such as name and level. In Algorithm 4, we had one token for each fragment in the Petri net. In Algorithm 5, we have many tokens for each fragment, one token for each agent. The token of an agent consists of its name, the set of its adjacent edges that are already computed as edges of the minimum spanning tree, its fragment name, and its fragment level. Algorithm 5 is operationally equal to Algorithm 4. Each action in Algorithm 5 is always a synchronous action of all agents of the fragment (or the two fragments) together. An agent cannot execute an action independently. We ensure this by transition guards.

The correctness of all refinement steps, we described so far, can be justified by well-known refinement techniques. The Petri net models of the Algorithms 4,5 and 6 use the same net as the model of Algorithm 3. During the refinements, we only change arc inscriptions, transition guards, and, in the step from Algorithm 4 to Algorithm 5, the place fragments was renamed into agents.

We now explain the first transition refinement step. In Algorithm 5, the merging of two fragments is a synchronous action of both fragments after they have sent connect-messages to each other. In Algorithm 6, this is refined by two concurrent actions of the fragments. Each fragment that receives a connect-message via an edge and has already sent a connect-message via this edge knows that both fragments will merge. Without further synchronisation with the other frag-

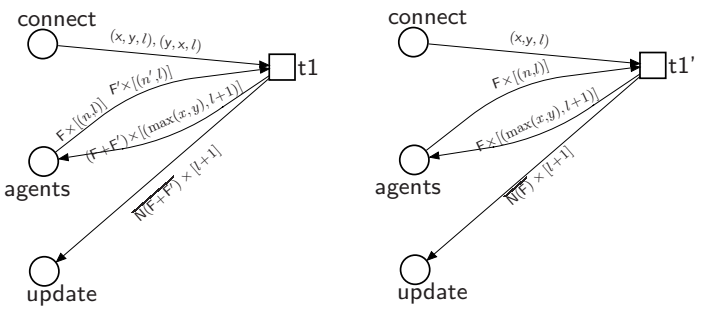


Fig. 7. Transition t1 and a replacement R for t1

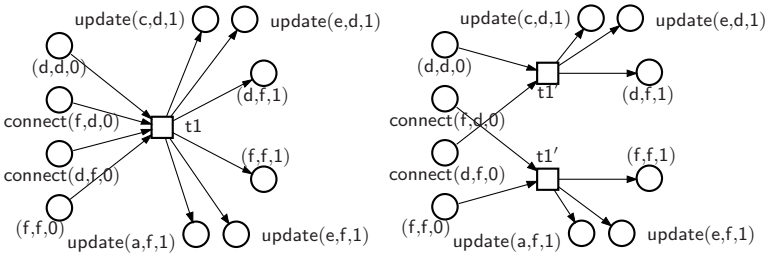


Fig. 8. The replacement of the merge transition in a concrete mode

ment it executes its part of the merging, i.e. each agent of the fragment gets the new name, and the level of the fragment is increased by 1. Furthermore, update-messages with the new name and level are sent to each neighbour. Technically this is done by replacing transition $t1$ in the model of Algorithm 6 by transition $t1'$ (see Fig. 7). Even though the replacement consists of only one transition, a run of the replacement will start with all preconditions for the old transition $t1$. Therefore each run of the replacement consists of two concurrent occurrences of $t1'$ in different modes.

Fig. 8 illustrates the merge transition of Algorithm 5 and its replacement in Algorithm 6 in a concrete mode. We consider again the example network in Fig. 2. We look at the two trivial fragments d and f. In Algorithm 5, they initially send connect-messages to each other because the edge with weight 4 is the minimal outgoing edge for both fragments. A connect-message (as well as an update-message) is given by a triple which indicates receiver, sender, and level of the sender's fragment. The trivial fragments are in the situation given on the left hand side: they use both connect-messages, and combine to form a new fragment and send update messages to their neighbours. An agent is given by a triple which indicates its name, its fragment's name (which is equal to its name for trivial fragments), and its fragment's level. In Algorithm 5, the information is already distributed to the agents, i.e. even after merging, there is a token for each agent. Both agents belong to the same fragment after firing of $t1$ because they have the same fragment name (f the maximum of d and f) and the same level.

On the right-hand side of Fig. 8, both agents fire $t1'$ in different modes. They use the connect-message from the other agent and send update-messages to their neighbours.

It is not trivial to prove that this replacement for $t1$ is indeed a transition refinement. The reason is that a merging action of a fragment may be in conflict with allowing another fragment with lower level to join. This kind of nondeterminism was the main problem for other attempts to prove the correctness the GHS algorithm (see subsequent discussion).

Let us investigate this situation a bit closer: we consider two fragments F_1 and F_2 that send connect-messages to each other and are therefore enabled to

perform the merge action independently of each other. The merge actions of both fragments become enabled simultaneously, namely when the last of both connect-messages is sent. Both fragments may also have received connect-messages from lower level fragments which enable them to perform join actions with these fragments. If F_1 received a connect-message from the lower level fragment F_3, it can decide first to let F_3 join and merge with F_2 afterwards or the other way round. No matter how it resolves the conflict or how F_2 resolves its conflict, we can show that the merge actions of both fragments are performed concurrently. This means that we can obtain each run of Algorithm 6 by expanding a run of Algorithm 5, such that each synchronous merge action of two fragments is replaced by two concurrent merge actions and furthermore, that each run of Algorithm 5 can be constructed like this. Therefore, this refinement step is a transition refinement.

The proof that the two merge actions are really performed concurrently is not trivial. It is given in [19]. We use the fact that the minimum spanning tree cannot contain a circle to deduce that the conflicts in both fragments are resolved independently, i.e. one join action has nothing to do with the join action in the other fragment.

Communication Inside a Fragment

With the next refinement step, we describe the communication inside a fragment.

If two fragments merged or joined, all agents of the fragment got the new name and level of the fragment at the same time. In Algorithm 7, both transitions, merge and join, are refined by a distributed broadcast algorithm. In a merge action, there is one distinguished edge that connects both fragments to a new fragment. In the refined merging of two fragments, the agents adjacent to this distinguished edge compute the new name and level of the fragment. From there, the information is spread like a wave over the rest of the new fragment by asynchronous messages. Not all existing communication channels are used for communication inside a fragment. Because a fragment is a subtree of the minimum spanning tree, all agents of a fragment are already connected by those communication channels, that are already computed to belong to the minimum spanning tree. Each agent knows which of its adjacent edges are already computed to have this property. We will call the neighbours on these edges the *tree neighbours* of the agent. We call all other neighbours of the agent its *basic neighbours* because the agent does not know at this stage if the channels leading to those agents are edges of the minimal spanning tree or not.

In Algorithm 7, a merging of two fragments works like this: A single agent x has enough information to start a merging if it receives a connect-message from a tree neighbour y. (In this case, x knows that it must have sent a connect-message to y before, otherwise y would be a basic neighbour.) The agent changes its fragment name to the maximum of x and y and increases the fragment level by 1. It sends an initiate-message with the new fragment name and level to all tree neighbours besides y and an update-message to all basic neighbours to inform them about the new level. An agent that receives an initiate-message, stores the

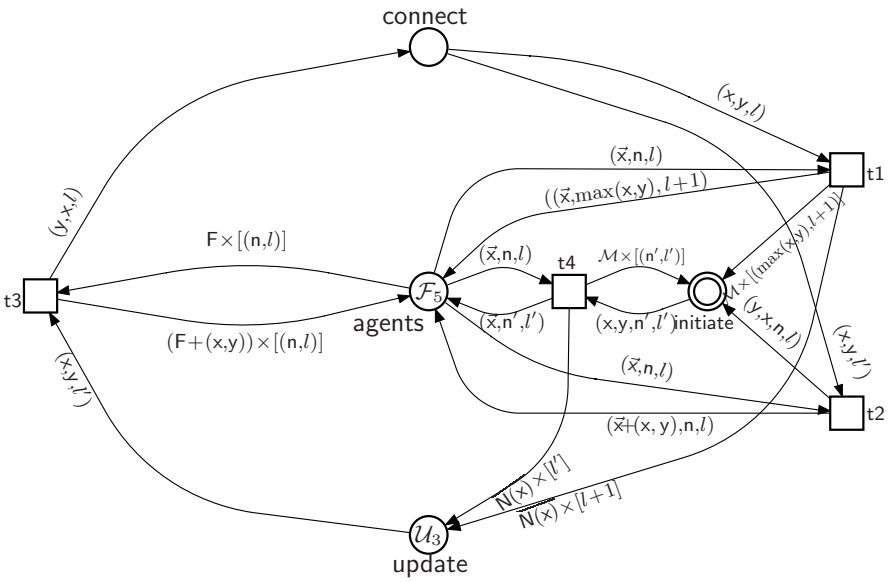


Fig. 9. Σ_7

new fragment name and level and sends further initiate-messages with this new information to all tree neighbours besides the sender of the initiate-message, and it sends update-messages with the new level to all basic neighbours.

A joining of a lower level fragment to a higher level fragment works similarly. An agent x that receives a connect-message from an agent y with a lower level than its own starts the joining. x sends an initiate-message with its own fragment name and level to y because the joining fragment takes over the name and level from the higher level fragment. When y receives the initiate-message from x it behaves as an agent receiving an initiate-message as described above. In this way the new information spreads as a wave over the lower level fragment as in the merging case.

In our Petri net model (given as illustration in Fig. 9 without sort definitions and transition guards), we have an extra place for each kind of message, one place for connect-messages, one place for update-messages, and one place for initiate-messages. Canonically, such places model unordered communication channels. Even though in the original GHS algorithm, FIFO channels are required, we use unordered channels and show with our proof that FIFO channels are in fact not necessary. The only exception are initiate-messages. Here we use a special place which models a FIFO channel for each pair of neighbouring agents. This FIFO behaviour may be implemented with unordered channels using message counters. If we use unordered channels for initiate-messages, the waves that distribute new information over the fragment could overtake each other. Imag-

ine the following situation: Fragment F_2 joins the higher level fragment F_1; a message wave informing all agents of the lower level fragment of the new name and level is initiated. In the meanwhile, F_1 merges with another fragment F_3. From this merge, a message wave is started with new information again. We now consider an agent of the former fragment F_3. It receives two initiate-messages, one about the joining and a more current one about the merging. From an unordered channel, the agent could chose to process the merge message first and the join message afterwards. The agent would actually decrease its level by receiving the join message after the merge message. This would result in an inconsistent system state and the algorithm would deadlock.

Actually, the algorithm works even if the initiate-messages are received unordered. We then have to require that an agent that receives an initiate-message with a lower level than its own just throws the message away. The agent knows that the message is out-of-date and it does not rename itself or spread the message. Unfortunately, we were not able to find a nice proof for the algorithm with unordered initiate-messages. Therefore, we use the FIFO requirement for initiate-messages.

This refinement step is a transition refinement. Each merge or join transition is refined by the distributed broadcast described above. We have to show again, that each started message wave is ended without interference of other actions of the algorithm and not altered or aborted early. The proof is again not trivial but very similar to the proof of the first transition refinement step. It involves the same mathematical arguments. Besides the transition refinement step from algorithm 5 to 6, this refinement step is the most important in the refinement process.

Joining and merging of fragments is now distributed to local actions of agents. Sending a connect-message is still a synchronous action of all agents of a fragment, and we did not describe *how* a fragment determines its minimal outgoing edge. We do this in six further refinement steps. Two of them are transition refinements. They are similar but even simpler than the transition refinements we have seen so far. The other four steps are again easy to prove with well-established methods such as simulation. Therefore, we will not discuss the next six refinement steps here.

For more details we refer the reader to the technical report [18], and the PhD thesis [19]. Fig. 10 shows the size of the final algorithm of the refinement hierarchy, the GHS algorithm.

7 Related Work

The proof of Chou and Gafni. In [4], Chou and Gafni propose a new proof technique: *stratified decomposition*. They investigate distributed algorithms which work in sequential phases from an abstract point of view. These phases usually operate concurrently in the actual distributed algorithm. They propose to divide the actions of such an algorithm into different layers. They define when such a stratified decomposition of an algorithm is serializable. Transposed into

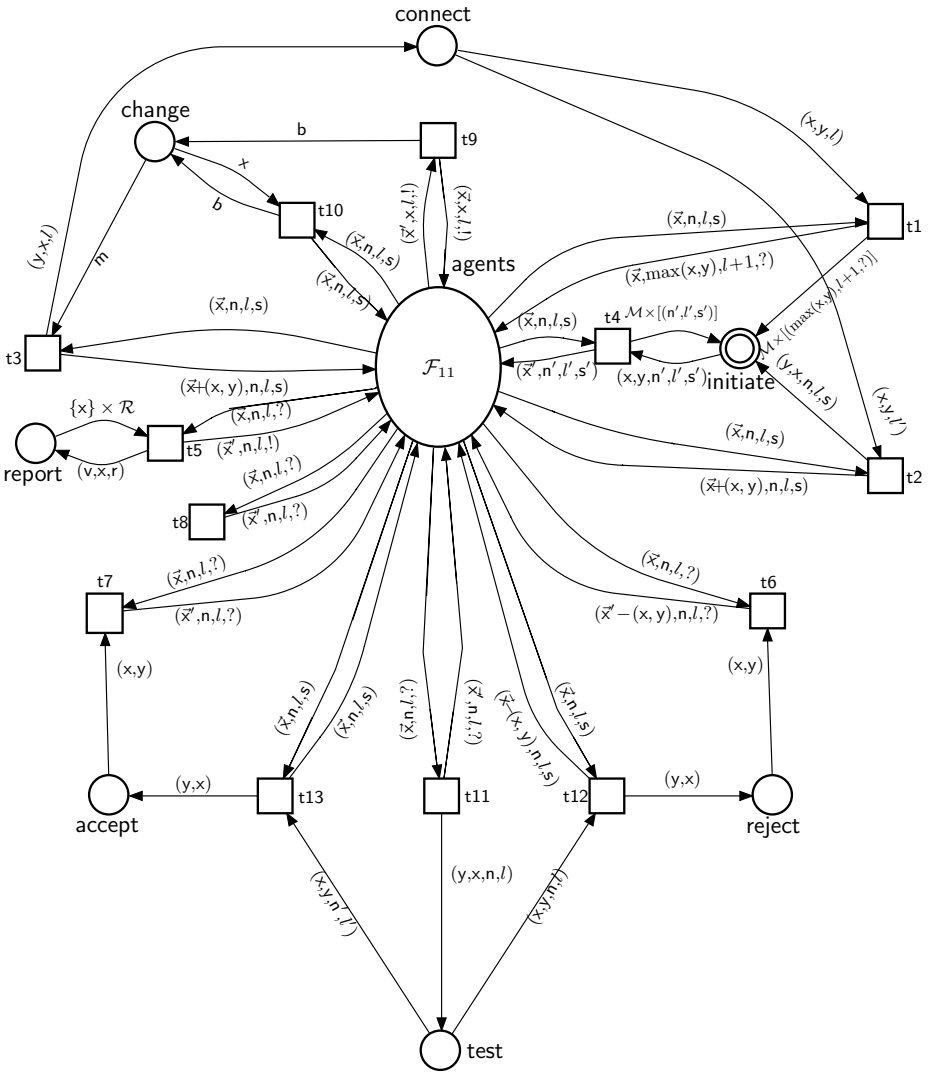


Fig. 10. The GHS algorithm

our formal model, this is the case if we can find cuts in each run of the algorithm such that between two consecutive cuts there are exactly the actions of one layer [15]. If the stratified decomposition of an algorithm is serializable then the algorithm yields the same results as the algorithm in which the layers are executed strictly sequentially. This work of Chou and Gafni is a development of the idea of *Communication Closed Layers* of Elrad and Francez [6].

Chou and Gafni prove the correctness of a simplified version of the GHS algorithm. In this version, it is deterministic when a lower-level fragment joins a higher-level fragment. There are no conflicts between merging and joining. By this exclusion of non-determinism, it is possible to find a stratified composition of the algorithm such that each fragment level corresponds to a layer. Stratified decomposition can not be used to prove the full GHS algorithm because the GHS algorithm is not serializable (see [18] for details).

The proof of Stomp and de Roever. Stomp and de Roever [22,23] also demonstrate the use of a new proof method: *sequentially phased reasoning*. This method is also a further development of communication closed layers. Similarly to stratified decomposition, this proof method can be applied to algorithms that execute concurrent phases that are executed in a sequential order from an abstract point of view.

Sequentially phased reasoning is more flexible than stratified decomposition. Unlike the layers in stratified decomposition, the phases in sequentially phased reasoning do not refer globally to the whole algorithm but only to groups of agents. Agents which participate in different phases run through these phases in the same order.

The complete proof of the GHS algorithm is given in the dissertation [21]. The GHS algorithm is divided into phases. The correctness of each phase is proved in isolation of the other phases. Then it is shown the phases in co-operation satisfy the principle of *sequentially phased reasoning*. The algorithm which is constructed of the phases is again the more deterministic version of the GHS algorithm. The last step of the proof introduces the non-determinism which leads to the full GHS algorithm.

The proof of Janssen and Zwiers. Janssen and Zwiers [12] want to solve the same problem as Chou, Gafni, Stomp and de Roever: understanding algorithms which execute concurrent phases that are executed in a sequential order from an abstract point of view. They chose an algebraic approach. In [12], they define a new program operator $\bullet$. This operator is called *layer composition* and is a hybrid of sequential and parallel program composition.

The layer composition $P \bullet Q$ of two programs P and Q is the program which executes independent actions of P and Q concurrently. In case of a conflict between an action of P and an action of Q, it executes the action of P.

The correctness proof of the GHS algorithm with layer composition is given in the dissertation [10]. In several refinement steps, again, the more deterministic version of the GHS algorithm is derived. Janssen explains that there should be one further big step from this version to the actual GHS algorithm which introduces the non-determinism in receiving connect-messages. Janssen states that the introduction of non-determinism in the refinement process is unusual. He cannot prove the correctness of the full GHS algorithm in his formal model.

The proof of Welch, Lamport and Lynch. Unlike the work [22,12,4] which is based on the concept of *Communication Closed Layers*, Welch, Lamport and

Lynch [25] present an entirely different approach to prove the correctness of the GHS algorithm.

They define a lattice of several specifications which starts with a simple initial specification and ends with the target specification GHS algorithm. They show that each specification in the lattice simulates its predecessor specification with respect to the safety property. Furthermore, they show that there is a path through the lattice such that each specification in the path simulates its predecessor specification in the path with respect to the liveness property. From this they can deduce the correctness of the GHS algorithm. The full proof is again given in a dissertation [24]. The proof is very technical and extensive. It uses more than 200 pages.

The proof of Hesselink. In [9], Hesselink describes the correctness proof of the GHS algorithm with the help of the theorem prover Nqthm. The proof is based on ghost variables, 166 invariants, and a decreasing variant function. 32 000 lines of code were entered for the proof.

8 Discussion of Our Proof and Conclusion

The proof given in this paper is inspired by [22,12,4]. Our refinement hierarchy differs in two fundamental points: In the first place, the non-determinism in a fragment joining a higher-level fragment is inherent to all algorithms of the hierarchy. Transition refinement is a method that can handle this kind of non-determinism. The number of occurrences of a (refined) transition and its causal order with other transitions may be non-deterministic. The second essential difference is the introduction of update-messages as auxiliary variables in the refinement process which are deleted in the last refinement step. The clear distinction between *communication between fragments* and *communication inside a fragment* is only possible thanks to the update-messages.

By using transition refinement, we were able to stepwise refine synchronous actions of several agents to distributed local actions of the agents. More than by other refinement concepts, the concept of transition refinement is inspired by *Communication Closed Layers* which were introduced by Elrad and Francez [6] and the work which developed this concept [26,23,11,4]. In these papers, algorithms are decomposed into phases which are executed sequentially from an abstract point of view. The order of the phases is deterministic and each phase is executed exactly once. In the actual algorithm the phases are partly executed concurrently.

Transition refinement can be seen as a generalisation of this concept. We can consider each transition of the system as a phase. If we refine one or more transitions of a system the replacements are executed concurrently to each other and to the rest of the system. Each concurrent run of the system has a sequentialization in which all phases are executed sequentially. However, the order of the phases may be different in different concurrent runs due to non-determinism in the system. In particular, it is in general non-deterministic how often a phase is executed or if it is executed at all.

Our proof shows that FIFO channels are in general not necessary for the correctness of this algorithm. We use unordered channels with the only requirement that initiate-messages from one agent to another agent must be received in the same order as they are sent. We explained that even this restriction can be removed but we did not give a formal proof for this.

Each formal proof is error prone. Hand-made proofs such as the one in this paper are prone to human error. Mechanised proofs are prone to errors in human input or compiler errors. To gain even more confidence in the correctness, our derived model of the GHS algorithm (Fig. 10) was implemented with the Petri Net Kernel [14] by researchers in the group of Wolfgang Reisig. Its correctness was tested (by model checking) for several fixed communication networks with up to 6 agents.

Acknowledgements

I would like to thank Hagen Völzer and the anonymous referees for their helpful comments. The research was funded by the Australian Research Council Large Grant A49937045: Effective Real-Time Program Analysis, and the German Research Foundation (DFG) funded project: Kompositionale Verifikation.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Comput. Sci.*, 82:253–284, 1991. previously: SRC Research Report 27, April 1988. 375, 376
2. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21:181–185, Oct. 1985. 382
3. L. Castellano, G. D. Michelis, and L. Pomello. Concurrency vs interleaving: an instructive example. *EATCS Bulletin*, 31:12–15, 1987. 375
4. C.-T. Chou and E. Gafni. Understanding and verifying distributed algorithms using stratified decomposition (extended abstract). In *Proceedings of the 7th Annual Symposium on Principles of Distributed Computing*, pages 44–65. ACM, 1988. 376, 388, 390, 391
5. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998. 375, 376, 380
6. T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, 1982. 389, 391
7. R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, Jan. 1983. 376, 377, 378
8. D. Gries. An exercise in proving parallel programs correct. *Communication of the ACM*, 20(12):921–930, 1977. 376
9. W. H. Hesselink. The incremental design of a distributed spanning tree algorithm. *Formal Aspects of Computing*, 11(E):(electronic archive), 1999. 376, 391

10. W. Janssen. *Layered Design of Parallel Systems*. PhD thesis, University Twente, Enschede, 1994. 390
11. W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91*, volume 527 of *LNCS*, pages 298–316, 1991. 391
12. W. Janssen and J. Zwiers. From sequential layers to distributed processes, deriving a distributed minimum weight spanning tree algorithm (extended abstract). In *Proceedings of the 11th Annual Symposium on Principles of Distributed Computing*, pages 215–227. ACM, 1992. 376, 390, 391
13. E. Kindler and H. Völzer. Algebraic nets with flexible arcs. *Theoretical Computer Science*, 262(1-2):285–310, 2001. 379, 381
14. E. Kindler and M. Weber. The petri net kernel. In K. H. Mortensen, editor, *Tool Demonstrations, ICAPTN 99*, pages 71–75, 1999. 392
15. S. Peuker. Phased decomposition of distributed algorithms. Informatik-Bericht 120, Humboldt-Universität zu Berlin, 1999. 389
16. S. Peuker. Property preserving transition refinement with concurrent runs: An example. In *International Conference on Application of Concurrency to System Design*, pages 77–86. IEEE Computer Society, 2001. 374, 375, 381, 382
17. S. Peuker. Concurrency based transition refinement for the verification of distributed algorithms. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication Based Systems*, LNCS. Springer, 2002. To appear. 375, 381
18. S. Peuker. Deriving a distributed minimum weight spanning tree algorithm with transition refinement for Algebraic Petri nets. Technical Report 02-10, Software Verification Research Centre, The University of Queensland, March 2002. 377, 388, 390
19. S. Peuker. *Halbordnungsbasierte Verfeinerung zur Verifikation verteilter Algorithmen*. PhD thesis, Humboldt-Universität zu Berlin, available via `http://dochost.rz.hu-berlin.de/abstract.php3/dissertationen/peuker-sibylle-2001-07-03`, 2001. 377, 382, 386, 388
20. W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, May 1991. 376, 379
21. F. A. Stomp. *Design and Verification of Distributed Network Algorithms: Foundations and Applications*. PhD thesis, Technische Universiteit Eindhoven, 1989. 390
22. F. A. Stomp and W. P. de Roever. A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract). In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 440–447. ACM, 1987. 376, 390, 391
23. F. A. Stomp and W. P. de Roever. Principles for sequential reasoning about distributed algorithms. *Formal Aspects of Computing*, 6E:1–70, 1994. 390, 391
24. J. L. Welch. *Topics in Distributed Computing*. PhD thesis, MIT/LCS/TM-361, 1988. 391
25. J. L. Welch, L. Lamport, and N. Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm (extended abstract). In *Proceedings of the 7th Annual Symposium on Principles of Distributed Computing*, pages 28–43. ACM, 1988. 376, 391
26. J. Zwiers. Compositional transformational design for concurrent systems. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference, Proc. of COMPOS '97*, volume 1536 of *LNCS*, pages 609–631. Springer-Verlag, 1998. 391

Token-Controlled Place Refinement in Hierarchical Petri Nets with Application to Active Document Workflow

David G. Stork[1] and Rob van Glabbeek[2]

[1] Ricoh California Research Center
2882 Sand Hill Road Suite 115, Menlo Park, CA 94025-7022
stork@rii.ricoh.com
[2] Department of Computer Science, Stanford University
Stanford, CA 94305
rvg@cs.stanford.edu

Abstract. We propose extensions to predicate/transition nets to allow tokens to carry both data and control information, where such control can refine special "refinable place nodes" in the net. These formal extensions find use in active document workflow, in which documents themselves specify portions of the overall processing within a workflow net. Our approach enables the workflow designer to specify which places of the target predicate/transition net may be refined and it enables the document author to specify how these places will be refined (via attachment of a token-generated "refinement net"). This apportionment of the overall task allows the workflow designer to set general constraints within which the document author can control the processing; it prevents conflicts between them in foreseeable practical cases. Refinable places are augmented with a permission structure specifying which document authors can refine that place and which document tokens can execute a node's refinement net. Our refined nets have a hierarchical structure which can be represented by bipartite trees.

1 Introduction

Document process workflow — such as the sequence of operations on documents in a loan application, employment application, insurance claim, purchase requisition, online credit verification at purchase, processing of patient medical records, distribution and sign-off on memos in an office, or editorial steps in the production of a magazine — is an increasingly important application of concurrency theory. While a number of ad-hoc high-level languages and commercial systems such as COSA, Visual Workflow, Forte Conductor, Verve Workflow, iFlow, InConcert, and SAP R/3 Workflow have been developed to serve such applications [16], there is nevertheless a need for a formal language that would allow workflow properties to be derived, for example halting, reachability, invariances, deadlock, and livelock. Further, if various high-level workflow languages can be expressed in a common formal language, their functional differences and

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 394–413, 2002.
© Springer-Verlag Berlin Heidelberg 2002

similarities can then be exposed and analyzed in that formal language. Finally, a unifying formal language would enable business partners to merge their workflows (for instance through chaining or synchronization) even though they use different high-level workflow languages [2].

Wil van der Aalst and his colleagues [1] have argued persuasively that Petri nets (of which predicate/transition nets form a subclass) provide such a formal foundation for document workflow for these and other reasons, specifically:

Formal Semantics: Petri net formalism provides precise definitions and clear semantics of both basic systems and those enhanced with attributes such as color, time, and hierarchy.

Graphical Nature: Petri nets have natural graphical representations and are thus intuitive, easy to learn, and admit natural human-machine interfaces supporting drag and drop, click and link, and other operations on icons representing processes and documents.

Expressiveness: Basic Petri nets can support the functional primitives needed to model existing document workflow systems.

Management Systems Can Be Modeled: Local states in Petri nets are represented explicitly and this allows for the modeling of implicit choices and milestones.

Reasoning about Properties: Petri-net-based process algebra rests on firm mathematical foundations, and this facilitates reasoning about network properties.

Analysis: A wealth of formal Petri net analysis techniques have been developed for proving properties (safety, invariance, deadlock, ...) and for calculating performance measures (response times, waiting times, occupation rates, ...).

Vendor Independence: Petri nets are a tool- and vendor-independent framework and as such they will not vanish amidst inevitable turmoil in the marketplace.

Rather than model or analyze existing workflows, in the below work we build upon and extend Petri nets to provide a foundation for more powerful, enhanced future workflow systems. While basic Petri nets are indeed attractive as a foundation for current workflow systems, Petri net theory as it stands is incapable of expressing properties we seek in such expanded workflows. In particular we explore the properties of *active document workflow* and introduce extensions to Petri net theory that enable them. In Section 2 we review the use of Petri nets in traditional document workflow, sketch what functionality we want in active document workflow, and mention analogies in other areas of computer science. Then, in Section 3 we review several Petri net and related formalisms and show that they are insufficient to express such active document workflow. In Section 4 we give the formal definition of token-controlled place refinement and we conclude in Section 5 with some future directions.

2 Document Workflow

2.1 Traditional Workflow

In traditional document workflow, a workflow designer specifies the processing steps to be applied to a (passive) document. If the workflow is implemented as a Petri net, then *tasks* or *transitions* (represented by squares) specify the operations to be performed, and *local states* or *places* (represented as circles) specify the status. The causal flow through the Petri net is indicated by arrows or *arcs* linking transitions to states and states to transitions (but never states to states or transitions to transitions). Documents are often represented as structureless tokens that flow through the network. States or conditions within the workflow — such as "copier tray empty," "return receipt sent," and so forth — are represented by place nodes, and the status is indicated by the presence or absence of structureless tokens.

Figure 1 shows a simple document workflow that might be used by a newspaper publisher. Ignore for the moment the bold circle, and consider how such a network implements a traditional workflow. Reporters in the field submit drafts of articles to the newspaper editorial office either by fax or by email; if an article is faxed, then the transmitted document is electronically scanned at the office to create an electronic version. The submission is logged, and then the electronic document is passed to an editor who makes corrections and passes the edited document to the typesetter.

2.2 Active Document Workflow

Consider now how the full network in Fig. 1 (i.e., with the bold circle) can represent *active* document workflow in the case of a single article submitted by Reporter 1. As before, Reporter 1 submits a draft article, but in this case the reporter has added special control instructions to the document. After the submission has been logged in the workflow network, this special control information is read and executed at the state node indicated by the bold circle. The control information in the document must be in a machine-readable form, and could specify tasks such as "translate paragraph 2 from French to English," "convert the entries in Table 5 from Japanese yen into U.S. dollars using last October's exchange rate," "send a copy of this document to every person in the international news division," or combinations of such operations. This processing is invoked much like function or library calls to a set of resources made available by the editorial office and known by the reporter. These are operations that for any number of reasons can be done more easily and rapidly in the office than in the field. For instance, the relevant extra information to be incorporated or processing software may reside behind a firewall of the editorial office; resources such as a translator or image processing systems may be available only within the office; and so forth.

In other applications, active documents might contain the following control information:

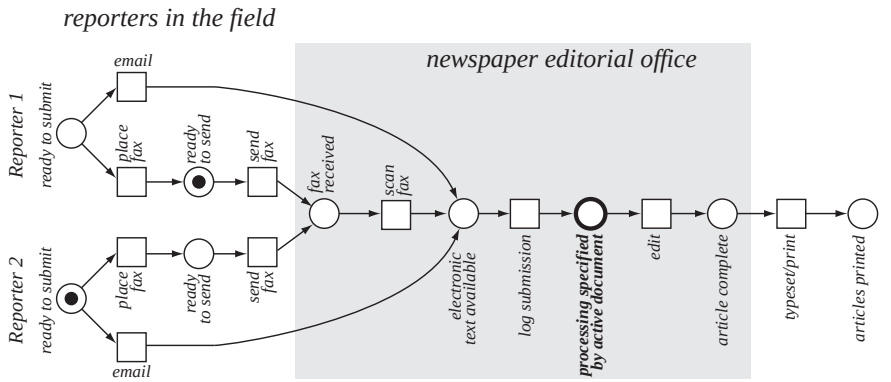


Fig. 1. A workflow implemented as a Petri net specifies the tasks (squares) and states (circles) linked by arcs. In this schematic example, two newspaper reporters can submit articles to the editorial office either by fax or by email, ultimately leading to typeset articles in the newspaper. In the current state, the draft of an article is being faxed by Reporter 1 (but has not yet been scanned); Reporter 2 is ready to submit an article. This state is shown by the dots (tokens), constituting a "marking." (Because this network has two input places, it differs slightly from workflow nets as defined in [1].) Active document workflow exploits processing in the bold circle (see Sect. 2.2)

- "send a return receipt to the sender"
- "print this letter on corporate letterhead and mail it to the following address"
- "crop photograph 3 to be square"
- "encrypt this document text using *PGP*"
- "determine the name of the recipient's corporate president and replace every occurrence of '`president`' in this document with his or her name"
- "change the text from one column to two column, except for tables and the appendix"
- "route this document to the CFO and request an acknowledgment; if an acknowledgment is received, pass the document to the next stage in the target workflow, otherwise encrypt the document and then pass the document to the next stage"
- "create a PDF duplicate of this document and send it to the internal document archive"

We stress that this enhancement is not merely the token-controlled selection of pre-existing operations anticipated by the workflow designer. Instead, it is the putting together of operations in an arbitrarily large number of unanticipated ways. Up to now, when workflow designers or users have specified such control tasks they have done so in an ad-hoc or informal way. For instance, in our example the reporter might have telephoned or sent a separate message to a secretary at the editorial office specifying the alterations to be performed on the submitted draft article.

The increase in electronic workflow — office intranets, networked office appliances, business-to-business nets, the world wide web — presents an opportunity for automating these processes and scaling them to large workflow systems. Our research goal is thus to provide a formal foundation for workflow that includes methods and protocols for automating the processing of active documents. Thus in the newspaper example, active document workflow would automate the processing and thereby ensure the editor and others receive a document in a form the author wishes, rapidly and with a minimum of direct human intervention. Standards for such active document workflow — specification in a formal language, conventions for calls to libraries of resources, acknowledgement protocols, and so on — would facilitate interorganizational active document workflows as well.

In our proposed extension to traditional workflow, the workflow designer specifies *which* places in the network can support token-specified processing, as well as the resources available for such processing; the document author specifies *how* those resources will be used for document processing. This apportionment of the overall document processing and routing allows flexibility to different applications. In fixed or rigid applications, the workflow designer retains absolute control and does not permit any document-controlled processes in the workflow network. That is, there would be no nodes such as the bold circle in Fig. 1, just as in traditional (passive) document workflow. In other applications, the workflow designer may permit some limited flexibility or freedom to the document author, perhaps only after important workflow processes have been completed, such as logging or archiving a copy of the original document. Moreover, the workflow designer provides a set of resources (libraries) and this set implicitly limits the operations that can be invoked by the document author. Finally, in networks serving multiple or highly variable operations, the workflow network can have multiple places where the author can specify processing, and provide large libraries of basic operations that can be invoked by the active document.

An example of rigid workflow is the processing of issued traffic tickets and fines at a police station or motor vehicle administration. Here the offender has no freedom in specifying how the ticket and fine are to be processed. An example in which modest freedom is granted to the document author is the newspaper example above. Consider now applications in which the document author has great freedom. Suppose a mobile professional has an office computer running an active document processing system. In this case the author might compose a business letter on a small portable device, send it electronically to his home computer that supports active document workflow. The document itself might specify that the letter is to be printed on corporate letterhead on special letter stock, be bound with the latest version of the corporate financial sheets resident on the home machine, and mailed to a particular set of recipients. Active document workflow of this type is particularly useful in low-bandwidth mobile or other non-synchronous applications. We can imagine, too, an outsource document processing service, where customers submit active documents specifying

typesetting, layout, translation, graphics processing, postal or electronic distribution, archiving, and notary services.

Our approach deliberately blurs and reduces the distinction between data and control, much as lambda expressions describe both functions and arguments in the λ-calculus and in the high-level programming languages based on the λ-calculus such as *Lisp* and *Scheme* [27,8]. In an indirect and informal way, file headers or extensions such as `.jpg` or `.gif` indicate that images are to be processed or rendered in a particular way, again in analogy to what we propose here. A stronger and richer analogy to our active document processing is the TEX typesetting and document preparation system, where a single source file contains a mixture of text and control functions that specify how that document should be formatted and the text rendered [15]. None of these systems, however, encompass the tasks of distribution and workflow more generally.

2.3 The Roles of Humans

Humans are essential components of any workflow system, of course, and we must clarify their responsibilities during design and use of our enhanced systems. An expert workflow designer constructs the workflow within the newspaper office, specifying the processing steps, flow relation, the refinable places and the library of functions that can be called by active documents. Similarly, this expert specifies the permission structures which will control which document authors can refine a given place (see Sect. 2.4). Document authors — here, the reporters — are non-experts and need only understand the functions available in the libraries. In this newspaper example, yet other humans serve as resources specified in the library. Thus the operation "translate from French to English" would send a document to a bilingual expert who needs no expertise in document workflow or other conventions.

2.4 Permission Structures

Consider next a workflow network within a corporation having both a legal department and a finance department. Documents such as letters, faxes or emails received by the corporation are classified and then routed through the corporation's document workflow to the appropriate department. Employees in such a corporation may wish to tailor ("reprogram") the workflow by means of documents themselves, even if they are outside the office. Surely, only members of the legal department should have ability and permission to alter the processing of documents that are routed to the legal department, and analogously for the finance department.

Our approach supports such alterations by allowing both temporary changes (affecting only the current document) and semi-permanent changes (affecting multiple, future documents) to the workflow. To this end we allow active documents not only to change (or refine) a workflow net, but also to undo or reverse such changes. The altered network is therefore hierarchical, where a network implementing the new functions lies at a deeper level than the original network. Our

formalism supports *permission structures* that specify who can make a change, and which documents are affected by such a change, as discussed below.

3 Related Formalisms

Given that our formalism for active document workflow will build upon Petri nets for the reason mentioned, there are nevertheless a number of related general formal approaches and specific proposals that must be considered. As we shall see, though, none of them have all the properties needed to enable the active document processing we envision.

Graph Rewriting Algebras: Graph rewriting algebras specify how to replace subgraphs with other subgraphs [6]. There is a structure governing these substitutions, such as formal composition, reversion (contraction), and so on. These algebras are inadequate for active document workflow because they contain no notion of the document itself controlling the rewriting.

Action Refinement: In concurrency theory and process algebra, action refinement refers to the substitution of complicated actions for simpler ones [4]. By itself, this technique is insufficient for active document workflow as there is no provision for these substitutions to take place during the execution of a process.

Petri Nets with Refinement: Refinement in a Petri net is the process by which a node in a network is replaced by other networks [24,11,7]. In standard Petri nets with refinement of places or transitions, however, the refinement is specified by the creator of the network rather than by information in tokens themselves. We shall employ token-controlled refinement as the mechanism for implementing the processing specified by a document author. Furthermore, our refinement need not *replace* an existing section of a net, but instead *attach* a refinement net in a way that will be described below.

Traditional Workflow Nets: Traditional workflow nets [1,2] represent documents by tokens, but in essence ignore the contents of those documents. For this reason they are not sufficiently expressive for token-controlled refinement. Furthermore, most effort on formal document workflow employs Petri nets such as basic or colored nets rather than the predicate/transition nets we shall require [3].

Petri Nets with Structured Tokens: In higher-level nets [23], such as predicate/transition nets [10,9] and colored Peri nets [13,14], tokens carry structured information that can be exploited at transitions. Higher-level net-based document workflow supports processing such as "if the document is a bill for over $10,000 from a familiar supplier, forward the bill to the Chief Financial Officer." Existing higher-level nets however do not allow changes to the workflow network, and hence no token-controlled changes to the workflow network in particular. To our knowledge, higher-level nets have never been used for refinement.

Reconfiguration: Dynamic reconfiguration, where processes are changed dynamically based on intermediate computed results, is used in a range of applications such as FPGA-based reconfigurable computing. The FPGA approach does not rest on formal foundations such as Petri nets, and the message packets in this approach bear only weak correspondence to the tokens in Petri nets for workflow [20]. Badouel and Oliver describe a Petri-net self-modifying or "reconfigurable net," which modifies its own structure by rewriting some of its components, but the reconfiguration information is not passed by the tokens [5]. Similar ideas appear already in the work of Valk [25].

Hierarchical Networks: In [26,18] hierarchical nets are studied in which tokens (objects) are nets as well, whose transitions may synchronize with the ones from the system net in which they travel. Although this approach is close to ours in a number of ways, it separates the behavior of the system net and the object net, and as of yet does not allow the passing of document contents from one to the other.

In summary, no previous formalisms support the full range of active document workflow and structures we envision. We now turn to the specification of such a formalism.

4 Token-Controlled Refinement in Hierarchical Nets

We begin with an informal description of basic token-controlled refinement, and subsequently develop, in various successive stages, formal definitions of the kind of nets that support such refinement. Because active documents will carry both data ("text") and control information ("control"), documents must be represented by *structured* tokens in a Petri net. If we consider the document as a file, the data and control may be intermixed so long as each is tagged and separable.

We call the traditional workflow net the "target net," to distinguish it from "refinement nets" specified by the structured token. In active document workflow, the workflow designer specifies the target net and any number of special "refinable places," which will serve as the loci of document-controlled operations. Figure 2 shows the internal structure of a refinable place, such as the bold circle in Fig. 1.

The top portion of Fig. 3 shows a target Petri net which includes a single refinable place. When a structured token enters the refinable place, refinement is then "enabled." Information in the token specifying the refinement net is read and the refinement net may then be attached between the input and output places i_s and o_s, as shown. In our approach, local *places* rather than transitions are refined because it simplifies formal definition and emphasizes that refinement amounts to the addition of extra tasks to a workflow process rather than a detailed explanation of how to execute an already specified process.

It is natural to consider the refinement net to lie at a different level than the target net, as shown in Fig. 3. Below we shall describe how another token carrying the instruction that a refinement should be undone may arrive, and the resulting *contraction* eliminates the refinement net.

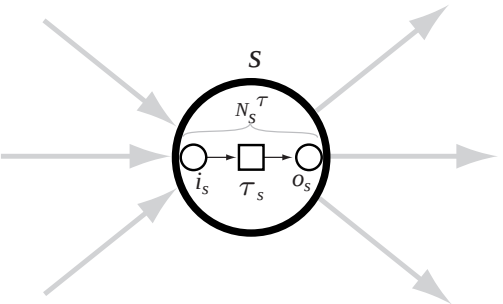


Fig. 2. A refinable place, $s \in S^R$, denoted by a bold circle, contains a formally simple "τ-net," N^τ, which consists of a unique input place, i_s, a unique output place, o_s, and transition τ_s, linked as shown. During token-controlled place refinement, i_s and o_s serve as "anchors" for the inserted refinement net, as shown in Fig. 3. The τ-transition represents a no-op transition and leaves the document unchanged. We use the symbol τ by analogy to τ-transitions in the calculus of communicating systems, CCS [19]. As we shall see, documents will pass through such a τ-transition either if there has been no refinement, or if the document does not have permission to execute a refinement net attached to this refinable place

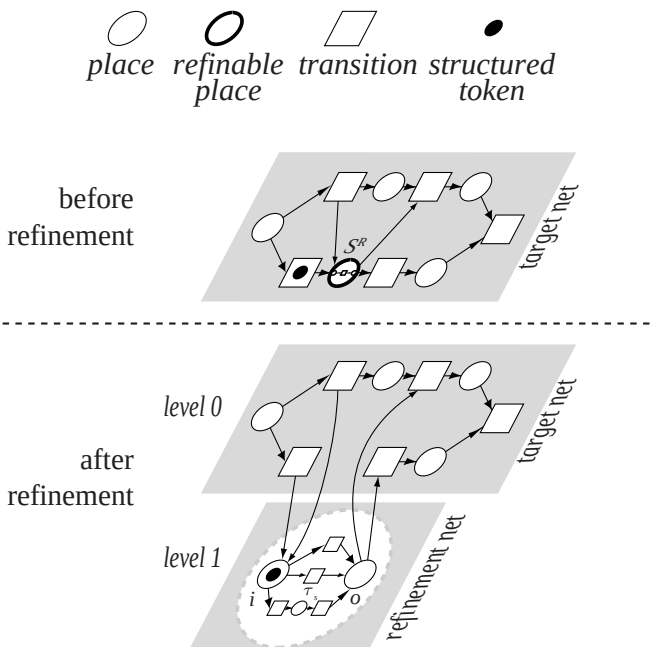


Fig. 3. Token-controlled process refinement occurs when a structured token encounters a refinable place. In this case, the control information in the token has been read, and implemented as a refinement net at the next level

Transitions in Workflow Nets Since we consider documents that contain control information, we must use net representations that employ structured tokens. In our systems, network transitions will modify documents, and to capture that fact we build upon the predicate/transition net formalism [10,9,23]. Figure 4 illustrates a transition in such a net where lower-case Roman letters (e.g., x and y) represent documents and upper-case Roman letters represent the operations performed on single or multiple documents. Thus, a Japanese translation of document y might be denoted $J(y)$, the concatenation of documents x and y might be denoted $x;y$, and so forth.

In practice, some of these operations are automatic, such as "encrypt this electronic document," while others require human intervention, such as "translate this document into Japanese." While we acknowledge that these operations may be quite difficult to implement, here we need not specify them beyond attributing a label to each so they could be listed in a library and called as needed.

While colored Petri nets [13,14] may have sufficient formal expressive power to serve as a foundation for our networks, colored nets are nevertheless awkward and unnatural for our needs because they obscure the structure of documents and their relationships. For instance, if two documents x and y are represented by tokens of differen colors, then the composite or concatenated document $x;y$ would be represented by a third color, thereby obscuring the document's com-

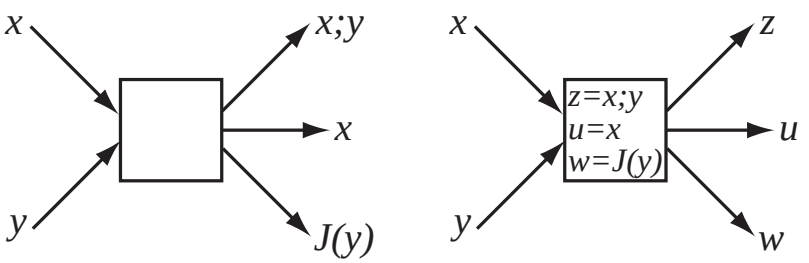


Fig. 4. Transitions in our workflow networks are based on those in traditional predicate/transition nets. In this example, documents x and y enter the transition. Operations are denoted by upper-case letters, with documents as arguments; a Japanese translation of y is denoted $J(y)$, the concatenation of the documents x and y is denoted by $x;y$, and so on, as shown on the left. Such processed documents — including ones subject to the *null* or "no-op" process, such as x above — are then emitted by the transition. A formally equivalent representation is to label the output arcs by symbolic names of the corresponding emerging documents, and write the transformations within the box, as shown on the right. This second representation emphasizes the fact that all information passing along an arc can be considered to be within a document. However, this second representation, as in colored Petri nets, obscures the structural information in documents, for instance that a particular document is a concatenation of two other documents

posite structure. In fact, giving an equivalent predicate/transition net is often the most efficient way of specifying a colored net.

Predicate/Transition Nets We let Σ be a *signature*, a list of names of operations f and predicates p on documents, each of which has an associated *arity*, $a(f), a(p) \in \mathbb{N}$. For instance, Σ could contain a binary operator ";" which is interpreted as a concatenation of two documents, or a unary operator $J(\cdot)$, interpreted as a Japanese translation of the document argument. It could further have a binary predicate $Q(\cdot,\cdot)$, which when evaluated to `True` says that its first argument is a document containing an endorsement for the statements contained in its second document, or a unary predicate $K(\cdot)$, which when evaluated to `True` says that its argument document has been signed by the president.

We also let $V = \{x, y, \ldots\}$ be a set of *variables* ranging over documents. Then $\mathbb{T}(V, \Sigma)$ denotes the set of *terms* over V and Σ, such as for instance $J(x); y$, the Japanese translation of document x concatenated with the document y. Furthermore, $\mathbb{F}(V, \Sigma)$ is the set of *formulas* over V and Σ in the language of first-order logic, such as $(\exists y.\ Q(y, x)) \vee K(x)$, saying that the statements in document x either have an endorsement, or have been signed by the president.

A *predicate/transition net* over V and Σ is given as a quadruple (S, T, F, λ) where S and T are disjoint sets of *places* and *transitions*, $F \subseteq (S \times V \times T) \cup (T \times V \times S)$ is the *flow relation*, and $\lambda : T \to \mathbb{F}(\Sigma, V)$ allocates to each transition a first-order formula over Σ and V called the *transition guard* [23].

The elements of S and T are graphically represented by circles and boxes, respectively, while an element $(p, x, q) \in F$ is represented as an *arc* from p to q, labeled with variable x. The formulas $\lambda(t)$ are written in the transition t. An arc $(s, x, t) \in F$ with $s \in S$, $x \in V$ and $t \in T$ indicates that upon firing the transition t, a document x is taken from place s. An arc $(t, y, s') \in F$ with $t \in T$, $y \in V$ and $s' \in S$ indicates that upon firing t a document y is deposited in place s'.

The transition guard $\lambda(t)$ selects properties of the input documents that have to be satisfied for the transition to fire, and simultaneously specifies the relation between the input and the output documents. The variables allocated to the arcs leading to or from t may occur free in the formula $\lambda(t)$. They provide the means to talk about input and output documents of this transition. Consider as an example a transition that consumes input documents x and y, and produces an output document z. The transition guard could be a formula that says that the transition may only fire if x is a PGP document that successfully decrypts with the key presented in document y; if these conditions are met, the decrypted document z is emitted.

Marked Predicate/Transition Nets and the Firing Rule Recall that Σ is a signature. A *Σ-algebra* is a domain $\mathcal{D}$ over which the operations and predicates of Σ are defined. In our case, of course, the elements of $\mathcal{D}$ denote documents. An *evaluation* $\xi : V \to \mathcal{D}$ over a Σ-algebra $\mathcal{D}$ assigns to every variable $x \in V$ a document $\xi(x) \in \mathcal{D}$. Such an evaluation extends in a straightforward manner to

terms and formulas over V and Σ, where $\xi(t) \in \mathcal{D}$ for terms $t \in \mathbb{T}(V, \Sigma)$, and $\xi(\varphi)$ evaluates to `True` or `False` for formulas $\varphi \in \mathbb{F}(V, \Sigma)$.

A *marking* of a predicate/transition net over V and Σ is an allocation of tokens to the places of the net. As these tokens are documents that are represented by elements in a Σ-algebra $\mathcal{D}$, we speak of a marking *over* $\mathcal{D}$. A *marked* predicate/transition net over V, Σ and $\mathcal{D}$ is a tuple (S, T, F, λ, M) in which (S, T, F, λ) is a predicate/transition net over V and Σ, and $M : S \to \mathbb{N}^{\mathcal{D}}$ a *marking*, which associates with every place $s \in S$ a multiset of documents from the Σ-algebra $\mathcal{D}$. The multiset $M(s) : \mathcal{D} \to \mathbb{N}$ is a function that tells for every possible document how many copies of it reside in that place.

For two markings M and M' over $\mathcal{D}$ we write $M \leq M'$ if $M(s)(d) \leq M'(s)(d)$ for all $s \in S$ and $d \in \mathcal{D}$. The marking $M + M' : S \to \mathbb{N}^{\mathcal{D}}$ is given by

$$(M + M')(s)(d) = M(s)(d) + M'(s)(d).$$

Thus, the addition of two markings yields the union of the respective multisets of tokens in each place in the net. The function $M - M' : S \to \mathbb{Z}^{\mathcal{D}}$ is given by $(M - M')(s)(d) = M(s)(d) - M'(s)(d)$; this function need not always yield a marking because it might specify a negative number of documents in a place. For a transition $t \in T$ in a predicate/transition net and an evaluation $\xi \colon V \to \mathcal{D}$, the *input* and *output* markings ${}^\bullet t[\xi]$ and $t[\xi]^\bullet$ of t under ξ are given by

$${}^\bullet t[\xi](s) = \{\!\{\xi(x) \mid (s, x, t) \in F,\ x \in V\}\!\}$$

and

$$t[\xi]^\bullet(s) = \{\!\{\xi(x) \mid (t, x, s) \in F,\ x \in V\}\!\}$$

for $s \in S$, in which $\{\!|, |\!\}$ are multiset brackets. A transition t is *enabled* under a marking M over $\mathcal{D}$ and an evaluation $\xi : V \to \mathcal{D}$, written $M[t, \xi\rangle$, if ${}^\bullet t[\xi] \leq M$. In that case t can *fire* under M and ξ, yielding the marking $M' = M - {}^\bullet t[\xi] + t[\xi]^\bullet$, written $M[t, \xi\rangle\, M'$.

Workflow Nets A *workflow net*, as defined by van der Aalst [2], is a Petri net (S, T, F) with two special places, i and o, that represent the input and output of the net. A workflow net does not have an initial marking. Instead, such a net acts on documents that are represented by tokens deposited by the environment in the input place i. Even if a net is finite and without loops, it may nevertheless represent an ongoing behavior, as the environment may continue to drop documents in the input place. The documents that arrive at the output place o are then carried away from there by the environment. Furthermore, i does not have incoming arcs (within the workflow net) and o does not have outgoing arcs (within the workflow net), and for every place $s \in S$ there should be a path in the net from i to o via s. Although in general it makes sense to consider multiple input and output places (as in Fig. 1), in the below discussions for simplicity we will follow van der Aalst in assuming just one of each.

Van der Aalst typically abstracts from the contents of documents by modeling them all as unstructured tokens. Here we give contents to documents by expanding place/transition nets to predicate/transition nets. Thus, a *predicate/transition workflow net* (over V and Σ) is a tuple (S, T, F, λ, i, o), and a *marked* predicate/transition workflow net (over V, Σ and $\mathcal{D}$) a tuple $(S, T, F, \lambda, i, o, M)$. The other parts of his definition are not affected.

An *initial marking* in a workflow net is a marking that puts only a single document in the input place of the net. In [2], workflow nets are often required to be *sound* in the sense that

- If an initial marking evolves into a marking that has a document in the output place, then there is only one document in the output place, and there are no documents left elsewhere in the net.
- If an initial marking can evolve into a marking M, then M can evolve further into a marking that has a token in the output place.
- There are no transitions in the net that can never be fired.

In van der Aalst [2], workflows are considered to be *case driven*, meaning that every case (started by a document token dropping into the input place) is executed in a fresh copy of the workflow net. This guarantees that documents in the workflow corresponding to different cases do not influence each other. Here we consider cases to be executed in parallel in the same workflow net, thereby creating the possibility for one case to influence the execution of another one. If we want the cases to be independent, we can augment each document token with a color or number, and require that transitions fire only when all incoming documents have the same color or number. Of course, each output then has the corresponding color or number. It is simple to implement the required bookkeeping tasks in predicate/transition nets.

Simple Hierarchical Workflow Nets In our formalization of token-controlled refinement, a workflow net may have one or more refinable places, thereby becoming a tuple $(S, T, S^R, F, \lambda, i, o)$ with $S^R \subseteq S - \{i, o\}$ the set of refinable places. The input and output place are not refinable. It may be helpful to think of a refinable place $s \in S^R$ as consisting of an input place i_s, an output place o_s, and a transition τ_s, as indicated in Fig. 2. All arcs leading to s go to i_s, whereas all arcs out of s go out of o_s. The transition τ_s has i_s as its only input place and o_s as its only output place. The τ_s transition does not have any observable effect and passes documents through unchanged. In fact, the behavior of any net up to branching bisimulation equivalence [12] is unaffected under substitution of a net as in Fig. 2 for any place s.

A *simple hierarchical workflow net* is a tuple $(S, T, S^R, F, \lambda, i, o, R)$ with (S, T, F, λ, i, o) a predicate/transition workflow net, $S^R \subseteq S - \{i, o\}$ a set of *refinable places*, and R a function that associates with every refinable place $s \in S^R$ a simple hierarchical workflow net $R(s)$.[1] The *refinement net* $R(s)$ is inserted in

[1] As the nets $R(s)$ are simple hierarchical workflow nets themselves, a formal definition involves recursion: The class of *simple hierarchical Petri nets* is the smallest class

the *top-level net* N at the place s: whenever a document arrives in s (i.e., in i_s), the document is transferred to the input place of the net $R(s)$, and then $R(s)$ runs concurrently with the top-level net. When a document reaches the output place of $R(s)$ it is transferred back to s (in fact to o_s), and can be used as input for transitions that need a document in place s in the top-level net.

A *marked* simple hierarchical workflow is a tuple $(S, T, S^R, F, \lambda, i, o, M, R)$ with (S, T, F, λ, i, o) a predicate/transition workflow net, $S^R \subseteq S - \{i, o\}$ a set of refinable places, $M : S \to \mathbb{N}^{\mathcal{D}}$ a marking of the top-level net, and R a function that associates with every refinable place $s \in S^R$ a marked simple hierarchical workflow net $R(s)$. Under this definition, a *marking* of a simple hierarchical workflow net has itself a layer at each level of the hierarchy; M is just the top layer, whereas the other layers reside in R. Transitions in such a net can fire at every level of the hierarchy, following the definitions for predicate/transition nets given before, except that tokens that arrive in refinable places end up one level lower in the hierarchy, and that tokens arriving in output places of a refinement net end up one level higher.

A non-hierarchical predicate/transition workflow net can be regarded as the simple case of a hierarchical workflow net where $S^R = \emptyset$. Classical notions of *place refinement* in Petri nets can be regarded as methods to flatten hierarchical nets into non-hierarchical ones. Such a flattening operation on workflow nets is extremely easy to define: just insert the net $R(s)$ at s by identifying the input place of $R(s)$ with i_s and the output place with o_s, while deleting the transition τ_s. As we will explore ways to change the hierarchical structure of nets dynamically (in particular by undoing a refinement during the execution of a net), it is important to clearly separate the refinement net associated to a refinable place from the top-level net. Therefore we will not flatten the net during refinement, but instead work with hierarchical nets. The operational behavior of a hierarchical net however can best be understood by picturing the flattened net.

Hierarchical Workflow Nets A *hierarchical workflow net* is defined just as a simple hierarchical workflow net above, except that the refinement function R has more structure. In particular, R associates to every refinable place not just a single refinement net, but a *list* of guarded refinement nets. Here a *guarded* net is a pair of a guard and a net, the guard being a first-order logic formula over V and Σ, with a distinguished variable x, which is the only variable that may occur free in that formula. When a structured token $d \in \mathcal{D}$ arrives in s, the guard of the first guarded net in the list is evaluated by taking x to be d. If the guard evaluates to `True`, the token descends to the input place of the corresponding net. Otherwise, the second guard is tested, and so on. The last element in the list is always the τ-net (cf., Fig. 2) with a guard that always evaluates to `True`. When a token arrives in the output place of any net in the list specified by $R(s)$, it moves upwards to s for further use in the higher-level net.

$\mathbb{H}$ of tuples $N = (S, T, S^R, F, \lambda, i, o, R)$ with (S, T, F, λ, i, o) a predicate/transition workflow net, $S^R \subseteq S - \{i, o\}$ a set of *refinable places*, and $R : S^R \to \mathbb{H}$.

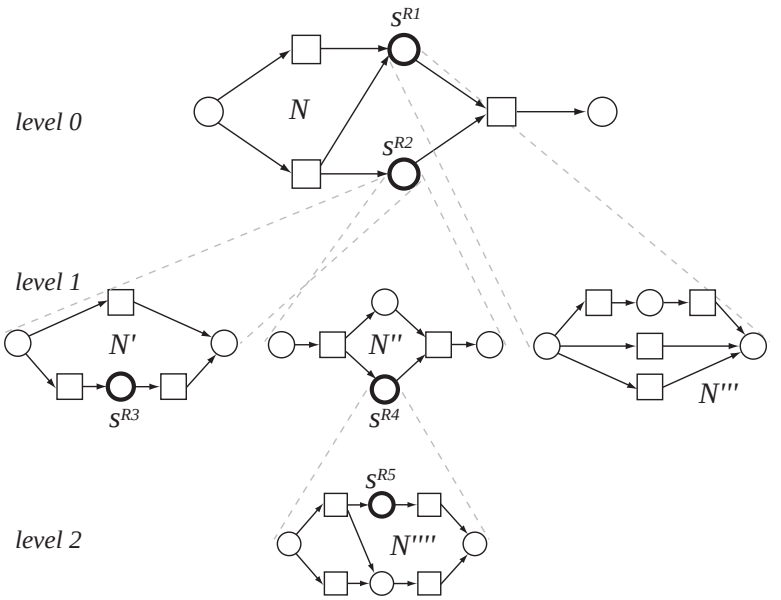


Fig. 5. Successive semi-permanent refinements lead to a hierarchical net. The target net N at the top has two refinable places. Here s^{R1} has been refined with the attachment of workflow net N''' and s^{R2} with both N' and N''. Finally, s^{R4} in N'' has been refined with N''''

Our aim in defining guards in this way stems from our recognition that a particular refinement may be useful only for documents of a particular type or from a particular author. The guards check in some way whether a document has "permission" to enter a particular refinement net. In case a document does not have permission to enter any meaningful refinement net associated to a certain place s, the structured token performs the τ-transition instead.

Figure 5 shows an example of a hierarchical net in which a single place has multiple refinement nets attached to it. The hierarchical structure of such a net can be represented by means of a bipartite tree, as indicated in Fig. 6. Each path from the root to a leaf consists of an alternating sequence of nets and refinable places.

Token-Controlled Refinement In order for our hierarchical workflow nets to allow token-controlled refinement, the refinement function $R(\cdot)$ will have even more structure than indicated above.

In active document workflow we allocate identifiers to refinable places, and documents carry instructions such as "when you land in a refinable place with this identifier, add the guarded net G to the list of guarded refinement nets." A refinable place s now carries an extra *refinement guard* $g_s(x)$ that checks whether

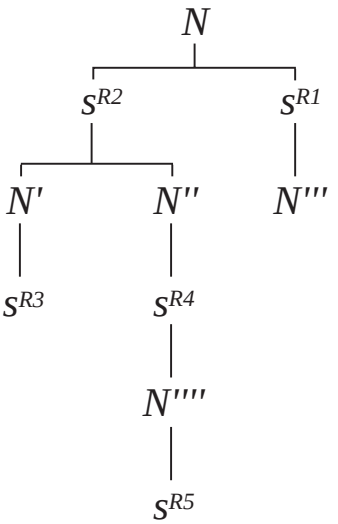


Fig. 6. The hierarchcial nets produced through token-controlled refinement, such as the one in Fig. 5, can be represented by a bipartite tree. In such a tree each path from the root, N, to a leaf consists of an alternating sequence of nets and refinable places

an incoming document has permission to initiate a refinement.[2] Moreover, it has a *refinement net extractor*, $ref_s(x)$, that reads the instruction pertaining to the refinement of s in a document x, and extracts the appropriate guarded refinement net from the document text. When, for an incoming document d, the refinement guard evaluates to `True` (i.e., $g_s(d)$ holds), and the refinement net extractor yields a guarded refinement net G (i.e. $ref_s(d) = G$), then this net is added to the head of the list in $R(s)$.

We also want to give documents the possibility to *remove* certain refinement nets. However, removal must not occur when that refinement net is still active, that is, when there is a token in the refinement net. Therefore we merely allow documents to change the entry guard of a refinement net to `False`, thereby preventing subsequent documents from entering that net. As soon as such a refinement net becomes inactive, in the sense that it can do no further steps, its role in the workflow comes to an end. We can eliminate idling "ghost nets" by incorporating a garbage collection process that regularly checks the workflow for inactive refinement nets with entry guards `False`, and removes them from the workflow.

In order to model token-controlled refinement fully, the R-component of a hierarchical net should have the structure $R(s) = (g_s(x), ref_s(x), L_s)$, where $g_s(x)$ is the refinement guard of s, $ref_s(x)$ is the refinement net extractor, and L_s is a

[2] To avoid confusion, the guards discussed above will henceforth be called *entry guards*.

list of *guarded refinement nets*, which are triples $(e(x), N, r(x))$ in which N is a refinement net, $e(x)$ is an *entry guard*, and $r(x)$ a *removal guard*. All guards are first-order logic formulas over V and Σ that have only the variable x occurring free. When a document d enters the refinable place s, first the removal guards of the guarded refinement nets in L_s are evaluated by substituting d for the free variable x. Each guarded refinement net $(e(x), N, r(x))$ for which $r(d)$ evaluates to `True` is earmarked for removal by assigning $e(x)$ to `False`. Next, the refinement guard is evaluated. In case $g_s(d)$ evaluates to `True`, the guarded refinement net $ref_s(d)$ is added to the head of the list L_s. Finally, the token d enters one of the refinement nets in L_s. To this end the token is evaluated by the entry guards one by one, and as soon as a guarded refinement net $(e(x), N, r(x))$ is encountered for which $e(d)$ evaluates to `True`, the token d is transferred to the input place of N.

In its initial state, any hierarchical workflow net has a refinement function R, such that for every refinable place s, the list L_s in $R(s)$ has only one guarded refinement net in it, namely $(\texttt{True}, N^\tau, \texttt{False})$. Because the removal guard of that τ-net always evaluates to `False`, that net will never be removed. Because that net's entry guard evaluates to `True`, this guarantees that every document will always succeed in entering the τ-net, should the document fail to enter any other refinement net.

The machinery above has been set up to facilitate semi-permanent token-controlled refinement. In order to achieve temporary refinement, in which only the token creating a refinement net may enter that net, the removal guard of the refinement net could be `True` (as well as the entry guard). This way, the next token that visits the refinable place will close that refinement net.

Summary We now summarize the formal definitions of the entities supporting active document workflow, based in part on traditional definitions and notations such as in [22,21].

Places: A set S (German, "stellen"), whose elements are indicated by circles.
Transitions: A set T (German, "transitionen"), whose elements are indicated by squares.
Variables: A set V of symbolic names, to be instantiated by documents.
Flow Relation: A relation $F \subseteq (S \times V \times T) \cup (T \times V \times S)$, indicated in a network by a set of directed arcs labeled with variables.
Input Place: The unique place i in a workflow net that accepts tokens (documents) from the environment. (In more general nets, there could be multiple input places.)
Output Place: The unique place o in a workflow net that emits tokens (documents) to the environment. (In more general nets, there could be multiple output places.)
Signature: A set Σ of names of n-ary operations and predicates on documents.
Terms: $\mathbb{T}$ over V and Σ.
Formulas: $\mathbb{F}$ over V and Σ, using the language of first-order logic.
Transition Guards: Formulas $\lambda(t)$ allocated to each transition t.

Predicate/Transition Workflow Net: A tuple (S, T, F, λ, i, o).
Tau Net: A net $N_s^\tau = (S_s^\tau, T_s^\tau, F_s^\tau, \lambda_s^\tau, i_s, o_s)$, where $S_s^\tau = \{i_s, o_s\}$, $T_s^\tau = \{\tau_s\}$, $F_s^\tau = \{(i_s, x, \tau_s), (\tau_s, x, o_s)\}$, and $\lambda_s^\tau(\tau_s) = \texttt{True}$.
Entry Guard: A formula $e(x)$ such that $e(d)$ tells whether document d has permission to enter a refinement net N.
Removal Guard: A formula $r(x)$ such that $r(d)$ tells whether document d has both the intention and the permission to remove a refinement net N.
Guarded Refinement Net: A triple $(e(x), N, r(x))$ consisting of a refinement net with an entry and a removal guard.
Guarded τ-Net: A tuple $(\texttt{True}, N_s^\tau, \texttt{False})$.
Refinable places: A set $S^R \subseteq S - \{i, o\}$, whose elements are indicated by bold circles.
Refinement Guard: A formula $g_s(x)$ such that $g_s(d)$ tells whether document d has permission to refine place s.
Refinement Net Extractor: A term $ref_s(x)$, such that $ref_s(d)$ extracts from document d the guarded refinement net that according to that document should be inserted at place s.
Refinement Function: A function R associates with each refinable place s a triple $R(s) = (g_s(x), ref_s(x), L_s)$, consisting of a refinement guard, a refinement net extractor, and a list L_s of guarded refinement nets, this list ending in the guarded τ-net.
Algebra of Tokens: An algebra based on the set $\mathcal{D} = \{d_1, \ldots\}$ of tokens equipped with the operators and predicates of Σ.
Marking: The assignment $M : S \to \mathbb{N}^{\mathcal{D}}$ of structured tokens to places in a net.
Marked Hierarchical Workflow Net: A net $N = (S, T, S^R, F, \lambda, i, o, M, R)$.

Transition Firing in Hierarchical Active Document Workflows Firing in networks supporting active document workflow is more complicated than in traditional workflows, of course, because permissions, refinement and contraction are supported.

The firing of a transition t now entails the following:

1. for an evaluation ξ of the variables that makes $\lambda(t)$ `True`
   (a) extract documents $\xi(x)$ from place s for every $(s, x, t) \in F$
   (b) deposit documents $\xi(y)$ in place s for every $(t, y, s) \in F$
2. when a structured token d enters a refinement place $s \in S^R$:
   (a) for each guarded refinement net $(e(x), N, r(x))$ in L_s, if $r(d) = \texttt{True}$: change $e(x)$ into `False`
   (b) evaluate $g_s(d)$. If $g_s(d) = \texttt{True}$ add guarded refinement net $ref_s(d)$ to the head of the list L_s in $R(s)$
   (c) go through the elements $(e(x), N, r(x))$ of L_s one by one until $e(d) = \texttt{True}$ and move d to the input place of N
3. when a structured token d enters an output of a guarded refinement net in $R(s)$
   (a) transfer that token one level up to s
   (b) continue executing the target net.

5 Future Directions

In summary, we have identified a new yet general and powerful operation, token-controlled refinement, and have proposed modifications to predicate/transition nets based on that operation support enhanced workflow. Our formal definitions of elementary properties lay a foundation for more sophisticated work, both in the theory of concurrency and workflow applications. Ideally, we would like to prove the preservation under refinement of properties such as workflow soundness, as put forth by van der Aalst [2]. Alas, in general predicate/transition nets his soundness property need not be preserved under refinement, and thus neither can this be the case for our extensions to predicate/transition nets, at least not without significant restrictions to the expressive power of our nets. Future work, then, will focus on properties such as the preservation of liveness through token-controlled refinement, much as has been shown for the composition of Petri nets [17]. These are steps toward an concurrent formalization of expanded workflow that should have important practical applications.

References

1. Wil M. P. van der Aalst. Three good reasons for using a Petri-net-based workflow management system. In *Information and process integration in enterprises: Rethinking documents*, pages 161–182. Kluwer Academic, Norwell, MA, 1998. 395, 397, 400
2. Wil M. P. van der Aalst. Interorganizational workflows: An approach based on message sequence charts and Petri nets. *Systems analysis – Modelling – Simulation*, 35(3):345–357, 1999. 395, 400, 405, 406, 412
3. Wil M. P. van der Aalst, Jörg Desel, and Andreas Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*. Springer, New York, NY, 2000. 400
4. Luca Aceto and Matthew Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, 1994. 400
5. Eric Badouel and Javier Oliver. Reconfigurable nets, a class of high level Petri nets supporting dynamic changes. *(WFM) Workshop within the 19th International Conference on Applications and Theory of Petri Nets*, pages 129–145, 1999. 401
6. Michel Bauderon and Bruno Courcelle. Graph expressions and graph rewriting. *Mathematical Systems Theory*, 20(83–127), 1987. 400
7. Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinement of Petri nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, number 483 in LNCS, 1991. 400
8. R. Kent Dybvig. *The Scheme programming language: ANSI Scheme*. Prentice Hall, Upper Saddle River, NJ, 1996. 399
9. Hartmann J. Genrich. Predicate/transition nets. In *Advances in Petri Nets 1986*, volume 254 of *LNCS*, pages 207–2471. Springer-Verlag, 1987. 400, 403
10. Hartmann J. Genrich and Kurt Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13(1):109–136, 1981. 400, 403
11. Rob van Glabbeek and Ursula Goltz. Refinement of actions in causality based models. In Jaco W. de Bakker, Willem Paul de Roever, and Grzegorz Rozenberg, editors, Proceedings REX Workshop on *Stepwise Refinement of Distributed Systems:*

Models, Formalism, Correctness, Mook, The Netherlands, May/June 1989, volume 430 of *Lecture Notes in Computer Science (LNCS)*, pages 267–300. Springer, 1990. 400

12. Rob van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996. 406
13. Kurt Jensen. Coloured Petri nets and the invariant-method. *Theoretical Computer Science*, 14(3):317–336, 1981. 400, 403
14. Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Vol. 1.* Springer–Verlag, 1992. 400, 403
15. Donald E. Knuth. *TEX: The Program, Computers and Typesetting.* Addison-Wesley, Reading, MA, 1986. 399
16. Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques.* Prentice Hall, Upper Saddle River, NJ, 2000. 394
17. Michael Köhler, Daniel Moldt, and Heiko Rölke. Liveness preserving composition of agent Petri nets. Technical report, Universität Hamburg, Fachbereich Informatik, 2001. 412
18. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the structure and behaviour of Petri net agents. In José-Manuel Colom and Maciej Koutny, editors, *Applications and Theory of Petri Nets 2001*, pages 224–241, 2001. 401
19. Robin Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, Cambridge, UK, 1999. 402
20. Ethan Mirsky and Andre DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In Peter Athanas and Kevin L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 157–166, 1996. 401
21. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 410
22. Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and analysis with Petri nets.* Springer, Berlin, Germany, 1998. 410
23. Einar Smith. Principles of high-level Petri nets. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri nets I: Basic models*, volume 1491 of *Advances in Petri nets*, pages 174–210. Springer, 1998. 400, 403, 404
24. Robert Valette. Analysis of Petri nets by stepwise refinements. *Journal of Computer and System Sciences*, 18(1):35–46, 1979. 400
25. Rüdiger Valk. Self-modifying nets, a natural extension of Petri nets. In *Proceedings ICALP '78*, volume 62 of *Lecture Notes in Computer Science (LNCS)*, pages 464–476. Springer, 1978. 401
26. Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manual Silva, editors, *Application and Theory of Petri Nets*, volume 1420 of *Lecture Notes in Computer Science (LNCS)*, pages 1–25. Springer, 1989. 401
27. Patrick Henry Winston and Berthold Klaus Paul Horn. *Lisp.* Addison-Wesley, Reading, MA, third edition, 1988. 399

Translating TPAL Specifications into Timed-Arc Petri Nets

Valentín Valero, Juan José Pardo, and Fernando Cuartero*

Departamento de Informática, Escuela Politécnica, Superior de Albacete
Universidad de Castilla-La Mancha
Campus Universitario s/n. 02071. Albacete, SPAIN
{Valentin.Valero,Juan.Pardo Fernando.Cuartero}@uclm.es

Abstract. It is well known that Petri nets are a very suitable model for the description and analysis of concurrent systems, and several timed extensions of PNs have been defined to specify the behaviour of systems including time restrictions. But some software designers still find it a little difficult to work directly with Petri nets, specially when dealing with large systems, and they prefer to work with a formalism closer to a programming language, such as algebraic specification languages. Our goal, then, in this paper is to present an automatic translation of specifications written in a (timed) algebraic language (TPAL) into a timed model of Petri nets (Timed-Arc Petri nets). In this task we try to exploit as far as possible the special characteristics of Timed-Arc Petri nets, more specifically the presence of *dead tokens*, which allows us to extend in some particular cases the classical translations.

Topics: Relationships between net theory and other approaches, Timed nets and timed process algebras.

1 Introduction

Process algebras and Petri nets have been widely used for the modelling and analysis of concurrent systems. Process algebras are much closer to a programming language, and thus, many software designers work gladly with them. In contrast to these, Petri nets have an easy interpretation, as they have a graphical nature. Furthermore, there is a solid mathematical foundation supporting them, and some techniques for the systematic analysis of properties are firmly established. Thus, some years ago it became evident that an integration of both formalisms would be very useful, because by doing so we could exploit the advantages of both description techniques. Goltz [10] presents a translation of CCS into Petri nets, while Taubner [18] defines such a translation for a more general process algebra, and more recently, a different approach has been introduced, the *Petri Box Calculus* [5], which takes the standard operators that we find in

* This work has been supported by the CICYT project "Performance Evaluation of Distributed Systems", TIC2000-0701-C02-02.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 414–433, 2002.
© Springer-Verlag Berlin Heidelberg 2002

process algebras, applying them to a domain of Petri nets (*Boxes*). One of the main advantages of this latter approach is that we get a compositional behaviour of the resulting nets.

Both formalisms, process algebras and Petri nets, have been extended to expand their application areas to those systems which exhibit a time-dependent behaviour that should be considered both in the modelling and the analysis process, such as distributed systems, communication systems and real-time systems. Many timed extensions of the classical process algebras have been proposed: timed CCS [22], temporal CCS [13], timed CSP [16] and TPAL [14]. We may also find several proposals for timed extensions of Petri nets. A survey of the different approaches to introduce time in Petri nets is presented in [7]. We can identify a first group of models, which assign time delays to transitions, either using a fixed and deterministic value [15,17,20] or choosing it from a probability distribution [4]. Other models use time intervals to establish the enabling times of transitions [12]. Finally, we also have some models that introduce time on tokens [1,2,6]. In such a case tokens become classified into two different classes: available and unavailable ones. Available tokens are those that can be immediately used for firing a transition, while unavailable tokens cannot. We have to wait a certain period of time for these tokens to become available, although it is also possible for a token to remain unavailable forever (such tokens are said to be *dead*). More recently, Cerone and Maggiolo-Schettini [8] have defined a very general model (statically timed Petri nets), where timing constraints are intervals statically associated with places, transitions and arcs. Thus, models with timing constraints attached only to places, transitions or arcs can be obtained by considering particular subclasses of this general framework.

Timed-Arc Petri nets [6,21,11,19,3] are a timed extension of Petri nets in which tokens have associated a non-negative real value indicating the elapsed time from its creation (*its age*), and arcs from places to transitions are also labelled by time intervals, which establish restrictions on the age of the tokens that can be used to fire the adjacent transitions. As a consequence of these restrictions some tokens may become *dead*, because they will never be available, since they are too old to fire any transitions in the future. The interpretation and use of Timed-Arc Petri nets can be obtained from a collection of processes interacting with one another according to a rendez-vous mechanism. Each process may execute either local or synchronization actions. Local actions are those that the process may execute without cooperation from another process, and thus in the Petri net model of the whole system they would appear as transitions with a single precondition place, while synchronization actions would have several precondition places, which correspond to the states at which each one of the involved processes is ready to execute the action. Then, each time interval establishes some timing restrictions related to a particular process (for instance the time that a local processing may require). In consequence, the firing of a synchronization action can be done in a time window, which depends on the age of the tokens on its precondition places.

Timed-Arc Petri nets are, therefore, a very appropriate model for the description of concurrent systems with time restrictions, such as manufacturing systems, real-time systems, process control, workflow systems, etc. Moreover, one of the main advantages of timed-arc Petri nets is that it is quite easy to get a timed-arc Petri net modelling a system that has been previously described by a timed process algebra. Then, one of the main goals of this paper is to show how this translation can be made, and how we can benefit from the token aging in this task.

We start, therefore, from an algebraic specification, written in TPAL [14], and we examine how to translate it into a timed-arc Petri net. In order to achieve this goal we have structured the paper as follows: in Section 2 we present timed-arc Petri nets and their semantics; in Section 3 both the syntax and the operational semantics of the timed process algebra (TPAL) are defined; and the translation of TPAL specifications into TAPNs is defined in Section 4. A complete example of a timed algebraic specification, and its corresponding TAPN is presented in Section 5, and finally, in Section 6 the conclusions and some indications about our future work are formulated.

2 Timed-Arc Petri Nets

Timed-arc Petri nets are a timed extension of Petri nets, which have their tokens annotated with an age (a real value indicating the elapsed time from its creation) and arcs connecting places with transitions have associated a time interval, which limits the age of the tokens to be consumed to fire the adjacent transition. In the particular model that we consider in this paper some transitions can be urgent, in the sense that no time can elapse once they are enabled.

Definition 1. (Labelled Timed-arc Petri nets)
Given a finite set of visible actions *Act*, we define a labelled timed-arc Petri net (LTAPN) as a tuple [1] $N = (S, T, F, times, \lambda)$, where S is a finite set of *places*, T is a finite set of transitions ($S \cap T = \emptyset$), F is the *flow relation* ($F \subseteq (S \times T) \cup (T \times S)$), *times* is a function that associates a closed time interval to each arc (s,t) in F, i.e.: $times : F|_{S \times T} \longrightarrow \mathbb{R}_0^+ \times (\mathbb{R}_0^+ \cup \{\infty\})$, and λ is the *labelling function*, $\lambda : T \longrightarrow Act \cup \{\tau\}$.

Thus, transitions can be labelled either with visible actions or with τ. These internal transitions (those labelled with τ) are considered urgent, i.e., no time can elapse when they are enabled, and they must be executed immediately, unless they are in conflict with some other transitions. Intuitively, these τ-transitions correspond to actions that the system can make internally, and therefore, once they are enabled it does not need to wait to execute these actions.

We will also use the standard notation to denote preconditions and postconditions of places and transitions:

$$^\bullet x = \{y \,|\, (y,x) \in F\} \qquad x^\bullet = \{y \,|\, (x,y) \in F\}$$

[1] We only consider arcs with weight 1 to simplify some definitions, but the extension to general arcs with greater weights is straightforward.

As previously mentioned, tokens are annotated with real values, so markings are defined by means of multisets on $\mathbb{R}_0^+$. More precisely, a marking M is a function:

$$M : S \longrightarrow \mathcal{B}(\mathbb{R}_0^+)$$

where $\mathcal{B}(\mathbb{R}_0^+)$ denotes the set of finite multisets of non-negative real numbers. Thus, as usual, each place is annotated with a certain number of tokens, but each one of them has associated a non-negative real number (its *age*). We will denote the set of markings of N by $\mathcal{M}(N)$, and using classical set notation, we will denote the number of tokens on a place s by $|M(s)|$.

As initial markings we only allow markings M such that for all s in S, and any $x > 0$ we have $M(s)(x) = 0$ (i.e., the *initial age* of any token is 0). Then, we define *marked labelled timed-arc Petri nets* (MLTAPN) as pairs (N, M), where N is a labelled timed-arc Petri net, and M is an initial marking on it. Given a MLTAPN (N, M) we define $Init(N) = \{s \in S \,|\, |M(s)| > 0\}$. As usual, from the initial marking we will obtain new markings, as the net evolves, either by firing transitions, or by time elapsing. A labelled timed-arc Petri net with an arbitrary marking can be graphically represented by extending the usual representation of P/T nets with the corresponding time information. In particular we will use the age of each token to represent it. Therefore, MLTAPNs have initially a finite collection of zero values labelling each place. □

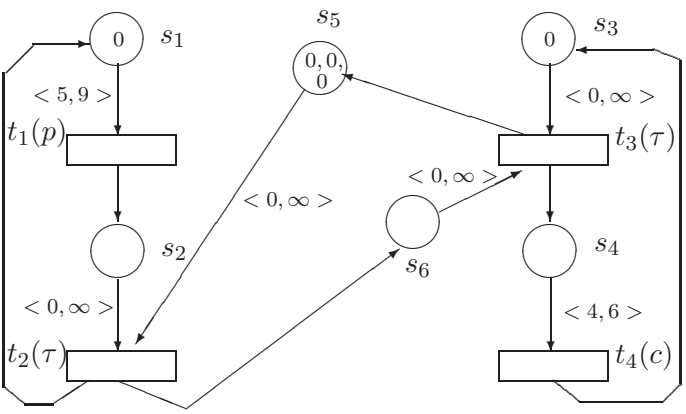


Fig. 1. Timed-arc Petri net modelling the PC-problem

In Fig. 1 we show a MLTAPN modelling a producer/consumer system, where we have represented by transition t_1 the action corresponding to the manufacturing process of the producer (p), which takes between 5 and 9 units of time, and by t_2 the action of including the generated object in the buffer (which is urgent). Notice that the initial tokens on s_5 represent the capacity of the buffer (3), and the arc connecting this place with t_2 is labelled by the interval $< 0, \infty >$,

because these tokens can be consumed at any instant in the future. Tokens on s_6 represent the objects on the buffer which have not been yet consumed. Transition t_3 models the (urgent) action of taking out an object from the buffer, which can occur at any instant. Finally, transition t_4 models the processing that makes the consumer for the objects extracted from the buffer (c), and this action takes between 4 and 6 units of time.

Let us observe that if the enabling time for the firing of one of these transitions (t_1 or t_4) expires, the system eventually becomes deadlocked, because we obtain a *dead* token either on s_1 or s_4.

Let us now see how we can fire transitions, and how we model the passage of time.

Definition 2. (Firing rule)
Let $N = (S, T, F, times, \lambda)$ be a LTAPN, M a marking on it, and $t \in T$.

(i) We say that t is *enabled* at the marking M if and only if:

$$\forall s \in {}^\bullet t \ \exists x_s \in \mathbb{R}_0^+ \text{ such that}$$

$$M(s)(x_s) > 0 \ \wedge \ x_s \in times(s,t)$$

i.e., on each precondition of t we have some token whose age belongs to $times(s,t)$. As usual, this will be denoted by $M[t\rangle$.

(ii) If t is enabled at M, it can be fired, and by its firing we reach a marking M', as follows:

$$M'(s) = M(s) - C^-(s,t) + C^+(t,s), \ \forall s \in S$$

where both the subtraction and the addition operators work on multisets, and:

– $C^-(s,t) = \begin{cases} \{x_s\} \text{ if } s \in {}^\bullet t, x_s \in times(s,t), \text{ and } M(s)(x_s) > 0 \\ \emptyset \quad \text{otherwise} \end{cases}$

– $C^+(t,s) = \begin{cases} \emptyset \quad \text{if } s \notin t^\bullet \\ \{0\} \text{ otherwise} \end{cases}$

Thus, from each precondition place of t we remove a token fulfilling (i), and we add a new token (with age 0) on each postcondition place of t. We denote these evolutions by $M[t\rangle M'$, but it is noteworthy that these evolutions are in general non-deterministic, because when we fire a transition t, some of its precondition places could hold several tokens with different ages that could be used to fire it. Besides, we see that the firing of transitions does not take up any time. Therefore to model the passage of time we need the function *age*, defined below. By applying it we age all the tokens of the net by the same time:

(iii) The function $age : \mathcal{M}(N) \times \mathbb{R}_0^+ \longrightarrow \mathcal{M}(N)$ is defined by:

$$age(M,x)(s)(y) = \begin{cases} M(s)(y-x) & \textit{if } \ y \geq x \\ 0 & \textit{otherwise} \end{cases}$$

We say that x units of time can elapse in (N, M) if and only if there is no τ-labelled transition enabled at $age(M, y)$, $\forall y \in [0, x]$. In this case, the marking obtained from M after x units of time without firing any transitions will be that given by $age(M, x)$.

□

Although we have defined the evolution by firing single transitions, this can be easily extended to the firing of *steps* or *bags* of transitions; those transitions that could be fired together in a single step could be also fired in sequence in any order, since no *aging* is produced by the firing of transitions. In this way we obtain step transitions that we denote by $M[R\rangle M'$. Finally, by alternating step transitions and time elapsing we can define a timed step semantics, where timed step sequences are those sequences $\sigma = M_0[R_1\rangle_{x_1} M_1 \dots M_{n-1}[R_n\rangle_{x_n} M_n$, where M_i are markings, R_i multisets of transitions and $x_i \in \mathbb{R}_0^+$, in such a way that $M_i[R_{i+1}\rangle M'_{i+1}$ and $M_{i+1} = age(M'_{i+1}, x_{i+1})$. Note that we allow $x_i = 0$ in order to capture the execution in time zero of two causally related steps.

Then, given a MLTAPN (N, M_0), we define $[M_0\rangle$ as the set of reachable markings on N starting from M_0, and we say that (N, M_0) is bounded if for every $s \in S$ there exists $n \in \mathbb{N}$ such that for all $M \in [M_0\rangle$ we have $|M(s)| \leq n$. A token on a place s at a marking M is said to be *dead* if it can never be used to fire any transitions, i.e., it will remain on its place forever, just growing up. Thus, we say that a marking is *dead* when all its tokens are *dead*.

We say that (N, M_0) is safe if $|M(s)| \leq 1,\ \forall s \in P$, for every $M \in [M_0\rangle$, and we will say that it is *soft-safe* if for every reachable marking we have on each place at most one non-dead token.

In a previous paper [19] we have shown that TAPNs without internal transitions (without urgency) have a greater expressiveness than PNs, even though TAPNs are not Turing complete, because they cannot correctly simulate a 2-counter machine. In that paper we proved that reachability is undecidable for this kind of TAPN. Other properties that we have studied in a more recent paper [9] are coverability, boundedness and detection of dead tokens, which are all decidable for non-urgent TAPNs. Decidability of coverability has been also proved in [3] for an extended version of non-urgent TAPNs, in which all arcs can be annotated with bags of intervals in $\mathbb{N} \times \mathbb{N} \cup \{\infty\}$.

However, it is very simple to show that LTAPNs with urgent transitions can simulate two-counter machines, and hence most of the properties become undecidable for them.

3 Syntax and Semantics of TPAL

In this section we briefly present both the syntax and semantics of TPAL, a timed process algebra which is very similar to some other proposals that we may find in the literature [13,16,22].

TPAL terms are those defined by the following BNF notation:

$$P ::= stop \mid a <x_1, x_2>; P \mid \tau; P \mid wait(x); P \mid P \,\square\, P \mid P \parallel_A P \mid P \backslash a \mid X \mid \mu X.P$$

where $A \subseteq Act$, $X \in Idf$ (set of identifiers), $a \in Act$, $x, x_1 \in \mathbb{R}_0^+$ and $x_2 \in \mathbb{R}_0^+ \cup \{\infty\}$.

The informal interpretation of the operators follows:

stop .- This represents a deadlock, i.e. no action can be executed (but time can elapse).

$a <x_1, x_2>; P$.- The action a can be executed after a delay in the time interval $[x_1, x_2]$ (with respect to the previous action). Once the action a has been executed, the process behaves as P. However, if we do not execute the action and the time expires, we will not be able to execute the action, and the process will have become deadlocked.

$\tau; P$.- The internal action τ is immediately executed, and the process behaves as P.

$wait(x); P$.- The process waits for x units of time, and then it behaves as $\tau; P$. A τ marks the end of the wait, and it guarantees that it finishes in exactly x units of time.

$P_1 \,\square\, P_2$.- This is essentially the classical external choice operator, but time can only elapse when both components are able to do that.

$P_1 \parallel_A P_2$.- This represents the parallel composition of P_1 and P_2 synchronizing on the actions in A. Again, time can elapse in a parallel composition only when both components can do that.

$P \backslash a$.- All instances of the action a in P are hidden, thus becoming urgent (τ).

$\mu X.P$.- This is the classical recursion operator, for the description of infinite behaviours.

Terms obtained from this syntax will be denoted by TPAL. We will usually restrict it to regular terms, for which terms $\mu X.P$ must be guarded (all appearances of X in P must be prefixed), and there is neither a parallel operator, nor a hiding operator in P. These regular terms will be denoted by RTPAL.

3.1 Operational Semantics

We define an operational semantics for the closed terms of TPAL by a labelled transition system in Plotkin and Milner's style (see Table 1). We have two kinds of transition rules:

$P \overset{x}{\longmapsto} P'$ The process P behaves as P' after x units of time, $x \in \mathbb{R}_0^+$.

$P \xrightarrow{e} P'$ The process P executes the action $e \in Act \cup \{\tau\}$ and then behaves as P' (no time elapses).

With rule R1 we capture that for the process stop the only possible evolution is the passage of time. Rules R2a-c capture the semantics of the timed prefix. Concretely, rule R2c states that once the time to execute an action has expired, we lose the ability to execute that action. Rules R3 and R4a-b capture the urgent character of internal actions, and the semantics of the wait operator. Rules R5a-b give the classical semantics for the external choice, while R7 states that the time can pass in an external choice only when both components are able to do that. Rules R8a-c are the classical ones for the parallel operator, while R8d is similar to R7. Semantics of the hiding operator is captured by rules R9a-c, the latter capturing the fact that once an action has been hidden, it becomes urgent. More precisely, the passage of x units of time for the process $P\backslash a$ is only possible if

Table 1. Operational Semantics of TPAL

R1) $\text{stop} \overset{}{\underset{x}{\longmapsto}} \text{stop} \quad \forall x \in \mathbb{R}_0^+$

R2a) $a < 0, x >; P \xrightarrow{a} P \quad \forall x \in \mathbb{R}_0^+$

R2b) $a < x_1, x_2 >; P \underset{x'}{\longmapsto} a < x_1 \dot{-} x', x_2 - x' >; P$
$0 \leq x' \leq x_2, \;\; x \dot{-} y = \max\{0, x - y\}$

R2c) $a < 0, 0 >; P \underset{x'}{\longmapsto} \text{stop} \quad x' > 0$

R3) $\tau; P \xrightarrow{\tau} P$

R4a) $\text{wait}(x); P \underset{x'}{\longmapsto} \text{wait}(x - x'); P\,, \; 0 \leq x' \leq x$

R4b) $\text{wait}(0); P \xrightarrow{\tau} P$

R5a) $\dfrac{P_1 \xrightarrow{e} P_1'}{P_1 \Box P_2 \xrightarrow{e} P_1'}$ **R5b)** $\dfrac{P_2 \xrightarrow{e} P_2'}{P_1 \Box P_2 \xrightarrow{e} P_2'}$

R6) $\dfrac{P_1 \underset{x_1}{\longmapsto} P_2 \quad P_2 \underset{x_2}{\longmapsto} P_3}{P_1 \underset{x_1 + x_2}{\longmapsto} P_3}$ **R7)** $\dfrac{P_1 \underset{x'}{\longmapsto} P_1' \quad P_2 \underset{x'}{\longmapsto} P_2'}{P_1 \Box P_2 \underset{x'}{\longmapsto} P_1' \Box P_2'}$

R8a) $\dfrac{P_1 \xrightarrow{e} P_1' \quad e \notin A}{P_1 \parallel_A P_2 \xrightarrow{e} P_1' \parallel_A P_2}$ **R8b)** $\dfrac{P_2 \xrightarrow{e} P_2' \quad e \notin A}{P_1 \parallel_A P_2 \xrightarrow{e} P_1 \parallel_A P_2'}$

R8c) $\dfrac{P_1 \xrightarrow{a} P_1', \quad P_2 \xrightarrow{a} P_2' \quad a \in A}{P_1 \parallel_A P_2 \xrightarrow{a} P_1' \parallel_A P_2'}$ **R8d)** $\dfrac{P_1 \underset{x'}{\longmapsto} P_1', \quad P_2 \underset{x'}{\longmapsto} P_2'}{P_1 \parallel_A P_2 \underset{x'}{\longmapsto} P_1' \parallel_A P_2'}$

R9a) $\dfrac{P \xrightarrow{e} P' \quad e \in Act \quad e \neq a}{P \backslash a \xrightarrow{e} P' \backslash a}$ **R9b)** $\dfrac{P \xrightarrow{a} P'}{P \backslash a \xrightarrow{\tau} P' \backslash a}$

R9c) $\dfrac{P \underset{x}{\longmapsto} P', \neg P \xrightarrow{a} \wedge \neg[\exists\, x' < x, \; P \underset{x'}{\longmapsto} P'' \,, \; P'' \xrightarrow{a}]}{P \backslash a \underset{x}{\longmapsto} P' \backslash a}$

R10a) $\dfrac{P\{\mu x.P/x\} \xrightarrow{e} P'}{\mu x.P \xrightarrow{e} P'}$ **R10b)** $\dfrac{P\{\mu x.P/x\} \underset{x}{\longmapsto} P'}{\mu x.P \underset{x}{\longmapsto} P'}$

there is no enabled a-action for all processes P' obtained from P by the passage of x' units of time, with $x' < x$. Finally, rules R10a-b capture the semantics for the recursion operator (the notation $P\{\mu X.P/X\}$ stands for the replacement of each occurrence of X in P by $\mu X.P$).

We will denote the labelled transition system that we may obtain by applying the rules in Table 1 for a process P by $lts(P)$.

4 Net Semantics for TPAL

In this section we define the translation of a TPAL process into a labelled timed-arc Petri net. We first consider the finite case, whithout recursion; the definition

is based on the syntactical structure of the given term. For the particular case of the external choice we provide two constructions, essentially depending on the initial parallelism of the components involved. Specifically, for the particular case of both components not having initial parallelism[2], the structure of the obtained net is very simple, and the procedure for doing this is well known. However, we also provide a definition for the external choice when one or even both components have initially a parallel behaviour. For the infinite case we have to deal with recursive terms, so we first present the translation for regular processes, showing later how we can go beyond these terms.

4.1 Finite Case

For the finite case we only consider processes without any recursion operators, these terms will be denoted by *fTPAL*.

Stop The *LTAPN* of *stop* only consists of a single place with a single token, whose initial age is 0:

$$N[\![stop]\!] = (\{s_0\}, \emptyset, \emptyset, \emptyset, \emptyset)$$
$$M_0(s_0) = \{0\}$$

Timed and Urgent Prefix Let $P = a <x_1, x_2>; P_1 \in fTPAL$, and $N_1 = ((S_1, T_1, F_1, times_1, \lambda_1), M_{1,0})$ the *MLTAPN* of P_1. The net of P is defined by including a new place, and a new transition, which is connected with every initial place of N_1:

$$N = ((S, T, F, times, \lambda), M_0)$$

where:

$$\begin{aligned}
S &= \{s_0\} \cup S_1\,, \;\; s_0 \notin S_1 \\
T &= \{t_0\} \cup T_1\,, \;\; t_0 \notin T_1 \\
F &= F_1 \cup \{(s_0, t_0)\} \cup \{(t_0, s_{1,0}) \mid s_{1,0} \in Init(N_1)\} \\
times(s,t) &= \begin{cases} times_1(s,t) \text{ if } (s,t) \in F_1 \\ (x_1, x_2) \quad\;\; \text{if } (s,t) = (s_0, t_0) \end{cases} \\
\lambda(t) &= \begin{cases} \lambda_1(t) \text{ if } t \in T_1 \\ a \quad\;\; \text{if } t = t_0 \end{cases} \\
M_0(s) &= \begin{cases} \{0\} \text{ if } s = s_0 \\ \emptyset \;\; \text{otherwise} \end{cases}
\end{aligned}$$

For the process $\tau; P$ the net has the same structure, the only changes being that now t_0 is labelled with τ, and the time interval of the arc (s_0, t_0) is $[0, 0]$. The net for the process $wait(x); P$ has got this structure too, but the time interval is now $[x, x]$.

[2] In the sense that neither of them has more than one initial place.

External Choice Let $P_1, P_2 \in fTPAL$ be two finite processes, and N_1, N_2 the nets associated with them, with $N_i = ((S_i, T_i, F_i, times_i, \lambda_i), M_{i,0})$, $i = 1, 2$, and $S_1 \cap S_2 = \emptyset$, $T_1 \cap T_2 = \emptyset$. For the definition of $N[\![P_1 \,\Box\, P_2]\!]$ we distinguish two cases, depending on the number of places initially marked in N_1 and N_2. If both components have only one initial place we may construct the net just by adding a new place, and removing the initial places of both nets; this new initial place is then connected with the postcondition transitions of the removed places. However, if at least one of the components involved has several initial places we need a construction which is a little more sophisticated, by adding some new places, each one corresponding to a pair of initial transitions of N_1 and N_2 respectively.

$$N[\![P_1 \,\Box\, P_2]\!] = \begin{cases} N_1 \Box_1 N_2 \text{ if } |Init(N_1)| = 1 \wedge |Init(N_2)| = 1 \\ N_1 \Box_2 N_2 \text{ otherwise} \end{cases}$$

where $\Box_1, \Box_2$ are both net operators, which are defined as follows:

- $N_1 \Box_1 N_2 = ((S, T, F, times, \lambda), M_0)$, where:

$$\begin{array}{l} S = \{s_0\} \cup S_1 \cup S_2 \setminus (Init(N_1) \cup Init(N_2)) \\ T = T_1 \cup T_2 \\ F = \{(s_0, t) \,|\, (s_i, t) \in F_i, s_i \in Init(N_i), i = 1, 2\} \cup \\ \qquad (F_1 \cup F_2) \setminus \{(s_i, t_i) \,|\, (s_i, t_i) \in F_i, s_i \in Init(N_i), i = 1, 2\} \\ times(s, t) = \begin{cases} times_i(s, t) \text{ if } (s, t) \in F_i,\ i = 1, 2 \\ times_i(s_i, t) \text{ if } s = s_0,\ s_i \in Init(N_i),\ (s_i, t) \in F_i, i = 1, 2 \end{cases} \\ \lambda(t) = \lambda_i(t), \text{ where } t \in T_i,\ i = 1, 2 \\ M(s) = \begin{cases} \emptyset \text{ if } s \neq s_0 \\ \{0\} \text{ if } s = s_0 \end{cases} \end{array}$$

- $N_1 \Box_2 N_2 = ((S, T, F, times, \lambda), M_0)$, where:

$$\begin{array}{l} S = S_{12} \cup S_1 \cup S_2, \text{ with } S_{12} = \{(t_{i_1}, t_{i_2}) \,|\, t_{i_1} \in Init(N_1)^\bullet,\ t_{i_2} \in Init(N_2)^\bullet\} \\ T = T_1 \cup T_2 \\ F = F_1 \cup F_2 \cup \{((t_{i_1}, t_{i_2}), t_{i_1}),\ ((t_{i_1}, t_{i_2}), t_{i_2}) \,|\, (t_{i_1}, t_{i_2}) \in S_{12}\} \\ times(s, t) = \begin{cases} times_i(s, t) & \text{if } (s, t) \in F_i,\ i = 1, 2 \\ (0, MW(P_1 \Box P_2)) & \text{if } s = (t_{i_1}, t_{i_2}) \in S_{12} \end{cases} \\ \lambda(t) = \lambda_i(t), \text{ where } t \in T_i,\ i = 1, 2 \\ M(s) = \begin{cases} \{0\} \text{ if } s = (t_{i_1}, t_{i_2}) \in S_{12} \vee (s \in Init(N_i), i = 1, 2) \\ \emptyset \text{ otherwise} \end{cases} \end{array}$$

where $MW(P)$ is defined as follows:

$$\begin{array}{ll} MW(stop) = 0 & MW(a < x_1, x_2 >; P) = x_2 \\ MW(\tau; P) = 0 & MW(wait(x); P) = x \\ MW(P_1 \,\Box\, P_2) = Max(MW(P_1), MW(P_2)) & MW(P \backslash a) = MW(P) \\ MW(P_1 \|_A P_2) = Max(MW(P_1), MW(P_2)) & \end{array}$$

This function gives us an upper bound for the maximum age that a non-dead token may have on any initial place (of course, it could also be infinite).

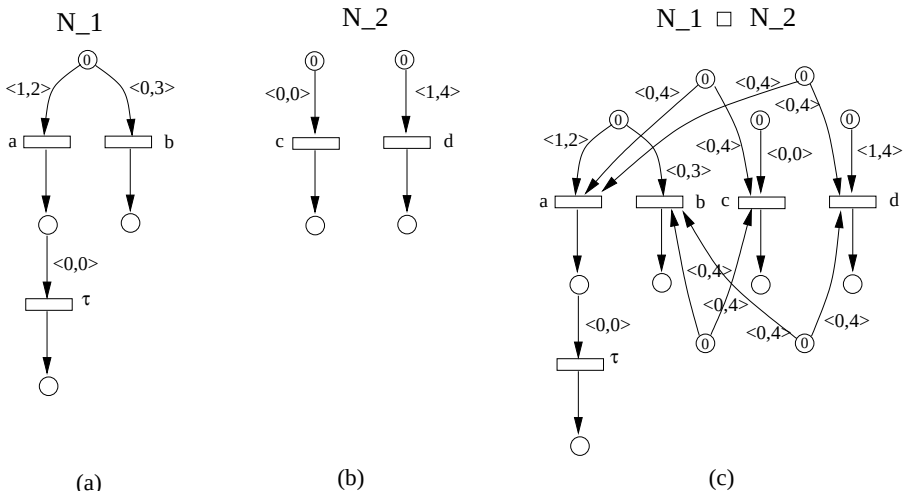


Fig. 2. Timed-Arc Petri nets of P_1, P_2 and $P_1 \square P_2$

Notice that $\square_1$ generates a net whose size is essentially the addition of the sizes of the argument nets (we just remove one place), while $\square_2$ increases the size, by including some new places and arcs. Specifically, there are $|Init(N_1)^\bullet| \times |Init(N_2)^\bullet|$ new initial places, and $2 \times |Init(N_1)^\bullet| \times |Init(N_2)^\bullet|$ new arcs.

Example 1. Let $P_1 = a <1,2>; \tau; stop \square b <0,3>; stop$, and $P_2 = c <0,0>; stop \|_\emptyset d <1,4>; stop$. The corresponding MLTAPNs for both processes are shown in Fig.2.a-b, and the net for $P_1 \square P_2$ is shown in Fig.2.c. □

Hiding Let P be a fTPAL process and N its corresponding MLTAPN. In order to construct the net associated with $P \backslash a$ we just need to change all the labels a in N by τ.

Parallel Operator Let $P_1, P_2 \in fTPAL$ be two finite processes, and N_1, N_2 the nets associated with them, with $N_i = ((S_i, T_i, F_i, times_i, \lambda_i), M_{i,0})$, $i = 1, 2$, and $S_1 \cap S_2 = \emptyset$, $T_1 \cap T_2 = \emptyset$. The net associated with $P_1 \|_A P_2$ is defined as follows:

$$N = ((S, T, F, times, \lambda), M_0)$$

where:

$$\begin{aligned}
S &= S_1 \cup S_2 \\
T &= ((T_1 \cup T_2) \setminus \{t_i \in T_i \,|\, \lambda_i(t_i) \in A,\ i = 1, 2\}) \cup \\
&\quad \{(t_1, t_2) \,|\, t_1 \in T_1, t_2 \in T_2, \lambda_1(t_1) = \lambda(t_2) \in A\} \\
F &= (F_1 \cup F_2)|_{(S \times T) \cup (T \times S)} \cup \{(s_1, (t_1, t_2)) \,|\, (s_1, t_1) \in F_1\} \cup \\
&\quad \{(s_2, (t_1, t_2)) \,|\, (s_2, t_2) \in F_2\} \cup \{((t_1, t_2), s_1) \,|\, (t_1, s_1) \in F_1\} \cup \\
&\quad \{((t_1, t_2), s_2) \,|\, (t_2, s_2) \in F_2\}
\end{aligned}$$

$$times(s,t) = \begin{cases} times_i(s,t) & \text{if } (s,t) \in F_i,\ i = 1,2 \\ times_1(s_1,t_1) & \text{if } t = (t_1,t_2) \wedge (s_1,t_1) \in F_1 \\ times_2(s_2,t_2) & \text{if } t = (t_1,t_2) \wedge (s_2,t_2) \in F_2 \end{cases}$$

$$\lambda(t) = \begin{cases} \lambda_i(t) & \text{if } t \in T_i \\ \lambda_1(t_1) & \text{if } t = (t_1,t_2) \end{cases}$$

$$M_0(s) = M_{i,0}(s) \text{ where } s \in S_i,\ i = 1,2$$

Thus, we join in a single transition all the pairs of transitions (t_1, t_2) which carry the same label in A, taking as preconditions (postconditions) of these new transitions the union of the preconditions (postconditions) of the two transitions involved. Consequently, with this definition the size of the generated nets will be in general a little smaller than the addition of the sizes of both argument nets.

Lemma 1. For any finite TPAL process P, $N[\![P]\!]$ is safe.

Proof: Immediate. □

We now state the equivalence between the operational semantics and the net semantics for finite processes. More specifically, the transition system associated with a finite TPAL process is *bisimilar* to the reachability graph that we may construct for the associated MLTAPN.

Theorem 1. Let $P \in fTPAL$ be a finite process and (N, M_0) its corresponding MLTAPN, according to the previous construction. Then, the labelled transition system associated with P, $lts(P)$, and the reachability graph of (N, M_0) are bisimilar, i.e., there is a relation $\rho \subseteq fTPAL \times \mathcal{M}(N)$ such that:

1. $(P, M_0) \in \rho$.
2. $\forall (Q, M) \in \rho$:
   (a) If $Q \overset{x}{\longmapsto} Q'$, then (N, M) can age by x units of time, and $(Q', age(M, x)) \in \rho$.
   (b) If $Q \xrightarrow{a} Q'$, then $\exists t \in T, \lambda(t) = a$, such that $M[t\rangle M'$, with $(Q', M') \in \rho$.
   (c) If (N, M) can age by x units of time, then there is a transition $Q \overset{x}{\longmapsto} Q'$, with $(Q', age(M, x)) \in \rho$.
   (d) If $M[t\rangle M'$, then there is a transition $Q \xrightarrow{\lambda(t)} Q'$, with $(Q', M') \in \rho$.

Proof: By structural induction, the base case (*stop*) is immediate. For the timed prefix, the urgent action prefix and the *wait* operator the reasoning is very similar, just adding some new elements (corresponding to the initial situation) to the relation ρ_1 that the induction hypothesis provides us. For instance, for $P = \tau; P_1$ we take $\rho = \{(\tau; P, M_0)\} \cup \{(Q, M) \mid M|_{Init(N[\![P]\!])} = \emptyset,\ (Q, M|_{S_1}) \in \rho_1\}$, where S_1 is the set of places of $N[\![P_1]\!]$.

For the external choice we have two constructions, so ρ must be defined in both cases. The definition for $\Box_1$ is quite easy, so we only present here the

definition corresponding to $\Box_2$, for which we take:

$$\begin{aligned}\rho = \{(P_1 \Box P_2, M_0)\} \cup \{(P_1' \Box P_2', age(M_0, x)) \mid P_1 \Box P_2 \overset{x}{\longmapsto} P_1' \Box P_2',\ x > 0\} \cup \\ \{(Q, M) \mid M|_{S_2 \setminus Init(N_2)} = \emptyset,\ (Q, M|_{S_1}) \in \rho_1,\ (\exists R_1 \subseteq Init(N_1)^\bullet, R_1 \neq \emptyset, \\ \forall t_1 \in R_1,\ \forall t_2 \in Init(N_2)^\bullet,\ M(t_1, t_2) = \emptyset) \wedge\ \exists x \geq 0,\ ((\forall s_2 \in Init(N_2), \\ M(s_2) = \{x\}) \wedge\ (\forall s \in Init(N[\![P]\!]) : (M(s) = \{x\} \vee M(s) = \emptyset)) \wedge \\ (\forall s_1 \in Init(N_1),\ M|_{S_1}(s_1) \neq \emptyset \Rightarrow \forall t_1 \in s_1^\bullet,\ \forall t_2 \in Init(N_2)^\bullet, \\ M(t_1, t_2) = \{x\}))\} \cup \\ \{(Q, M) \mid M|_{S_1 \setminus Init(N_1)} = \emptyset,\ (Q, M|_{S_2}) \in \rho_2,\ (\exists R_2 \subseteq Init(N_2)^\bullet, R_2 \neq \emptyset, \\ \forall t_2 \in R_2,\ \forall t_1 \in Init(N_1)^\bullet,\ M(t_1, t_2) = \emptyset) \wedge\ \exists x \geq 0,\ ((\forall s_1 \in Init(N_1), \\ M(s_1) = \{x\}) \wedge (\forall s \in Init(N[\![P]\!]) : (M(s) = \{x\} \vee M(s) = \emptyset)) \wedge \\ (\forall s_2 \in Init(N_2),\ M|_{S_2}(s_2) \neq \emptyset \Rightarrow \forall t_2 \in s_2^\bullet,\ \forall t_1 \in Init(N_1)^\bullet, \\ M(t_1, t_2) = \{x\}))\}\end{aligned}$$

where $N_i = N[\![P_i]\!]$, and S_i is the set of places of $N[\![P_i]\!]$. Notice that once we have fired a first transition, let us say t_1 of N_1, all places (t_1, t_{2_j}) become unmarked, which disables all the initial transitions of N_2, but we may still fire the same initial transitions of N_1 that could be fired taking this net isolated.

For the parallel operator, ρ can be defined as follows:

$$\rho = \{(Q_1 \|_A Q_2, M) \mid (Q_1, M|_{S_1}) \in \rho_1,\ (Q_2, M|_{S_2}) \in \rho_2\}$$

where S_i is the set of places of $N[\![P_i]\!]$, $i = 1, 2$.

And finally, for $P = P_1 \backslash a$ we take $\rho = \{(Q \backslash a, M) \mid (Q, M) \in \rho_1\}$ □

4.2 Infinite Case

We now consider recursive processes of *RTPAL*. Some changes must be made in the definition of the external choice, since for this construction to work properly we require initial places (for both components) not to have any incoming arcs, which can be achieved by *unfolding* the outer recursion that could appear on the external choice components.

Then, $N[\![P_1 \Box P_2]\!] =_{def} N[\![Unfold(P_1)]\!] \Box_i N[\![Unfold(P_2)]\!]$, where either $\Box_1$ or $\Box_2$ are applied according to the same criteria as in the finite case. *Unfold* is defined as follows:

$$\begin{aligned}&Unfold(stop) = stop && Unfold(a <x_1, x_2>; P) = a <x_1, x_2>; P \\ &Unfold(\tau; P) = \tau; P && Unfold(wait(x); P) = wait(x); P \\ &Unfold(P \backslash a) = Unfold(P) \backslash a && Unfold(\mu X.P) = Unfold(P\{\mu X.P/X\}) \\ &Unfold(P_1 \,\Box\, P_2) = Unfold(P_1) \,\Box\, Unfold(P_2) \\ &Unfold(P_1 \,\|_A\, P_2) = Unfold(P_1) \,\|_A\, Unfold(P_2)\end{aligned}$$

Notice that this definition is correct, since we are only dealing with regular processes. When the unfolding is required, the size of the net obtained will be significantly larger than the addition of the sizes of the argument nets, due to the replication of places and transitions.

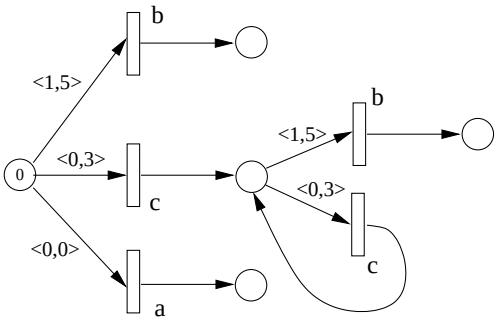


Fig. 3. MLTAPN for the process of example 2

Recursion In order to construct the MLTAPN associated with a regular process $\mu X.P$ we follow two steps:

1. We first construct the net corresponding to $P(X)$, taking for the identifier X a single place as we did for the process *stop*. Notice that as we are only dealing with regular terms, we will not find any terms such as $X \backslash a$, $X \Box Q$ or $X \|_A Q$. Besides, these X-places will have only a single precondition transition, since every appearance of X must be prefixed. We will denote the set of X-places by S^X.
2. Then, we remove these X-places, and we connect their precondition transitions to every initial place of the net (we take $Init(N[\![\mu X.P]\!]) = Init(N[\![P(X)]\!])$).

Example 2. Let $P = a <0,0>; stop \Box \mu X.(b <1,5>; stop \Box c <0,3>; X)$. After the *unfolding* the second component of the outer external choice becomes:

$$b < 1,5 >; stop \Box (c < 0,3 >; \mu X.(b <1,5>; stop \Box c <0,3>; X))$$

Then, by applying the previous definitions we obtain the MLTAPN shown in Fig.3. □

Lemma 2. For any RTPAL process P, $N[\![P]\!]$ is safe.

Proof: This is a consequence of P being regular. Notice that if an external choice appears within a recursion, both components must be sequential, i.e., only $\Box_1$ will be applicable in this case. This observation is important, because the application of $\Box_2$ within a recursion would be problematic, since some new places are created with $\Box_2$, and some of these new places retain some tokens even after the resolution of the conflict. □

Theorem 2. Let $P \in RTPAL$ be a regular process and (N, M_0) its corresponding MLTAPN, according to the previous construction. Then, the labelled transition system associated with P, $lts(P)$, and the reachability graph of (N, M_0) are bisimilar.

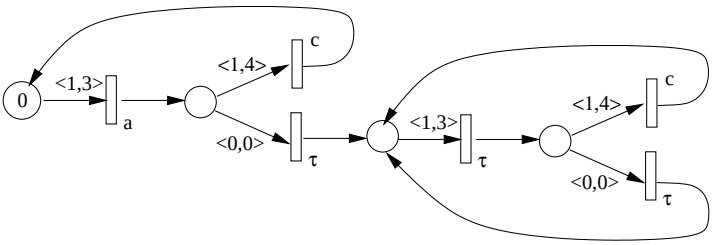


Fig. 4. Translation of a hiding within a recursion

Proof: By structural induction. We only need to explain how to construct ρ for the recursion, since the reasoning for the other operators coincides with that made for the finite case.

Let $P = \mu X.Q_1$, and ρ_1 the bisimulation that we may obtain for $Q_1(X)$ by applying the induction hypothesis (taking the net associated with it, according to the first step of the construction). Then, we can take:

$$\rho = \{(\mu X.Q_1, M_0)\} \cup \{(Q\{\mu X.Q_1/X\}, M|_{S_1 \setminus S_1^X}) \,|\, (Q, M) \in \rho_1, Q \neq X\}$$

where S_1 is the set of places of $N[\![Q_1(X)]\!]$ and S_1^X is the set of X-places in S_1. Notice that only one X-place can become marked in $N[\![Q_1(X)]\!]$ at the same time, because P is regular; besides, no other place in $N[\![Q_1(X)]\!]$ will be marked in that case. □

4.3 Beyond Regular Terms

The previous construction for recursive terms works fine for regular terms, but we would like to extend the translation a bit further. A first restriction that can be lifted with some minor changes is the banning of a hiding operator within a recursion. In this case, when we generate the net for $P(X)$ we obtain some (X, A)-places, which are essentially X-places, but all the actions in the set A are hidden for X. Therefore, we can *replicate* $N[\![P(X)]\!]$ for every (X, A)-place, hiding in these nets the actions in A, and then linking the preconditions of each (X, A)-place with the initial places of the corresponding net. Figure 4 shows the MLTAPN corresponding to the process $\mu X.(a < 1, 3 >; ((c < 1, 4 >; X) \Box ((a < 0, 0 >; X) \backslash a)))$.

Lifting the restriction concerning parallelism within recursion is not possible in general, because we would generate infinite nets. However, in some particular cases we may also apply our construction for recursive processes with parallel subterms; consider for instance the process $P = \mu X.(a < 5, 8 >; X \parallel_\emptyset b < 1, 2 > ; c < 0, 1 >; stop)$. By the time that the left-hand side component cycles, we must have the righ-thand side either finished or *dead*. In consequence, the net that we associate to this process (see Fig.5) will not be safe, but *soft-safe*, because some *dead tokens* will remain on the subnet corresponding to the righ-thand side component when we put new tokens on the initial places.

Therefore, we can extend our construction for those recursive terms $\mu X.P$ fulfilling the following conditions:

1. There is no recursive subterm in P.
2. Every occurrence of X in P is prefixed.
3. No hiding [3] is allowed in P.
4. For each subterm $P_1 \|_A P_2$ in P, X only appears in one of these components, either P_1 or P_2. Besides, the *maximum time* for the other component to finish its execution must be finite and smaller than the addition of the *minimum time* for this component (where X appears) to cycle and the *minimum time* to reach that subterm $P_1 \|_A P_2$ from P, i.e., assuming that X appears in P_1:

$$Max_time(P_2) < Min_time(P_1, X) + Min_time(P, P_1 \|_A P_2)$$

   where *Max_time* and *Min_time* can be defined as follows:
   - $Max_time(stop) = 0$
     $Max_time(a < x_1, x_2 >; P_1) = x_2 + Max_time(P_1)$
     $Max_time(\tau; P_1) = Max_time(P_1)$
     $Max_time(wait(x); P_1) = x + Max_time(P_1)$
     $Max_time(P_1 \Box P_2) = Max(Max_time(P_1), Max_time(P_2))$
     $Max_time(P_1 \|_A P_2) = Max(Max_time(P_1), Max_time(P_2))$
     $Max_time(P \backslash a) = Max_time(P)$
   - $Min_time(Q, Q) = 0$
     In all following cases: $Q \neq$ first argument (on left-hand side of definition)
     $Min_time(stop, Q) = \infty$
     $Min_time(X, Q) = \infty$
     $Min_time(a < x_1, x_2 >; P_1, Q) = x_1 + Min_time(P_1, Q)$
     $Min_time(\tau; P_1, Q) = Min_time(P_1, Q)$
     $Min_time(wait(x); P_1, Q) = x + Min_time(P_1, Q)$
     $Min_time(P_1 \Box P_2, Q) = Min(Min_time(P_1, Q), Min_time(P_2, Q))$
     $Min_time(P_1 \|_A P_2, Q) = Min(Min_time(P_1, Q), Min_time(P_2, Q))$
     $Min_time(P \backslash a, Q) = Min_time(P, Q)$
5. Every external choice $P_1 \Box P_2$ of type 2 in P satisfies:

$$MW(P_1 \Box P_2) < Min_time(P_1 \Box P_2, X) + Min_time(P, P_1 \Box P_2)$$

In our previous example $Max_time(P_2) = 3$, $Min_time(P_1, X) = 5$, and $Min_time(P, P_1 \|_\emptyset P_2) = 0$.

Lemma 3. Let $P \in fTPAL$ be with $Max_time(P) < \infty$. Then, any reachable marking of $N[\![P]\!]$ in a time greater than $Max_time(P)$ will be dead.

Proof: It can be easily checked by structural induction that $Max_time(P)$ is an upper bound for P to stop. Thus, this is an immediate consequence of Theorem 1. □

[3] This restriction is considered again for simplicity, but it could be lifted by finite replication.

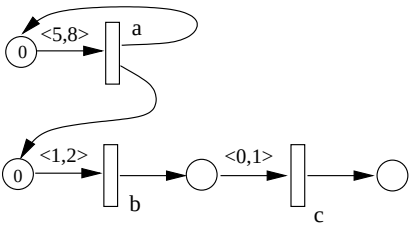


Fig. 5. $N[\![P = \mu X.(a < 5, 8 >; X \|_{\emptyset} b < 1, 2 >; c < 0, 1 >; stop)]\!]$

Lemma 4. For any TPAL process P fulfilling the conditions 1-5, $N[\![P]\!]$ is soft-safe.

Proof: We can reduce the proof to recursive processes fulfilling the conditions 1-5. It can be checked by structural induction that $Min_time(P, Q)$ is really a lower bound for the time that P requires to reach the subterm Q. By construction, the only net operators for which some unused tokens can remain on the corresponding subnet when we reach a X-place are $\Box_2$ and $\|_A$:

- With $\Box_2$ we have created some new places (t_1, t_2), and some of them remain marked once the choice has been resolved. Nevertheless, the arcs leaving these places are labelled by $< 0, MW(P_1 \Box P_2) >$, and thus, these tokens will be dead by the time that a new collection of tokens can arrive at these places, because of condition 5. The remaining tokens on these initial places of the subnet corresponding to $P_1 \Box P_2$ will be dead too, because their arcs must be labelled with values $< x_1, x_2 >$, with $x_2 \leq MW(P_1 \Box P_2)$.
- Let $P_1 \|_A P_2$ be a subterm of P, with X in P_1. Then, no token remaining on the places of the subnet $N[\![P_2]\!]$ can be alive when a collection of new tokens arrives at the initial places of the subnet $N[\![P_1 \|_A P_2]\!]$, because of condition 4 and Lemma 3. A finite number of parallel operators can appear in P_1, so we proceed by induction on that number:
  - No parallel operator appears in P_1: in this case the only remaining tokens in $N[\![P_1]\!]$ after a cycle can be those corresponding to an external choice of type 2, for which we have already proved the *soft-safeness*.
  - We now assume that n parallel operators appear in P_1, i.e., we have a subterm $P_{11} \|_B P_{12}$ in P_1, where n_1 parallel operators appear in P_{11} and n_2 in P_{12}, with $n - 1 = n_1 + n_2$. It follows by condition 4 that X can only appear in either P_{11} or P_{12}, so let us assume that it appears in P_{11}. Then, by condition 4 and Lemma 3 we may conclude again the *soft-safeness* for the places of the subnet corresponding to P_{12}, and by the induction hypothesis we conclude the *soft-safeness* for the places of $N[\![P_{11}]\!]$.

□

Corollary 1. Let $P \in TPAL$ be a process for which every subterm $\mu X.Q$ fulfills the conditions 1-5. Then, the reachability graph of $N[\![P]\!]$ and the labelled transition system associated with P are bisimilar. □

5 Example: Train-Gate Controller

As an illustration, we consider an example of an automatic controller that opens and closes a gate at a railroad crossing. The specification consists of three processes: $TRAIN$, $GATE$ and $CONTROL$, which have the following behaviour:

$$
\begin{aligned}
TRAIN &= \mu X.circulating <10,20>; approach <0,0>; in <5,5>; crossing <4,4>; \\
&\quad exit <0,0>; X \\
GATE &= \mu Y.lower <0,\infty>; down <1,2>; raise <0,\infty>; up <1,3>; Y \\
CONTROL &= \mu Z.approach <0,\infty>; reaction <1,1>; lower <0,0>; exit <0,\infty>; \\
&\quad reaction <1,1>; raise <0,0>; Z \\
TGC &= ((TRAIN \|_{\{approach,exit\}} CONTROL) \|_{\{raise,lower\}} GATE) \backslash approach \backslash exit \\
&\quad \backslash reaction \backslash lower \backslash raise
\end{aligned}
$$

The *train* sends an *approach* signal to the controller 5 minutes before it enters the crossing (*in*). It takes 4 minutes to cross, and then it sends the *exit* signal to the controller. The controller needs 1 minute to respond to the *approach* signal, and then it orders the closing of the gate (*lower*). It also needs 1 minute to respond to the *exit* signal, raising the gate in this case. The *gate* takes between 1 and 2 minutes to close, and from 1 to 3 minutes to open.

The behaviour of the complete system is described by the process TGC, in which some actions are hidden, because we want to enforce their execution when they are enabled.

The corresponding MLTAPN for this system, obtained by applying the translation, is depicted in Fig. 6.

6 Conclusions

We have defined a translation of a timed process algebra (TPAL) into timed-arc Petri nets with urgent transitions. This translation has been firstly defined for regular terms of TPAL, but we have also seen how to extend this translation for non-regular terms that fulfill certain conditions. In this latter case we have benefited from the token aging, obtaining *soft-safe* nets, for which every reachable marking has on each place at most one non-dead token.

Our future work on this subject will focus on the analysis of the generated nets, firstly studying techniques of reduction, and then going onto the verification of properties on these nets. We are also developing a tool supporting TPAL, which currently includes a translation of TPAL specifications into a kind of dynamic state graph, so part of our future work will also be to implement the translation into TAPNs in that tool, as well as the simulation and verification of the generated timed-arc Petri nets.

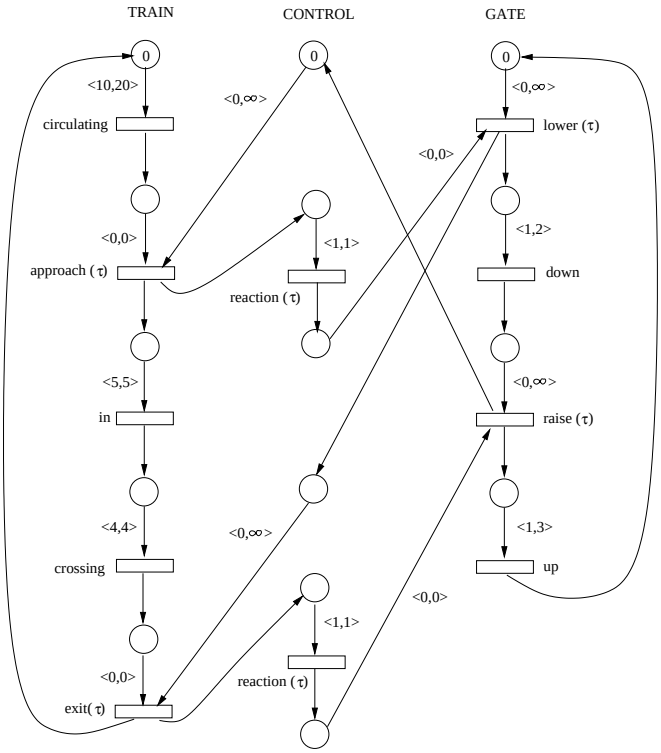


Fig. 6. MLTAPN modelling the Train-Gate Controller

Acknowledgement

The authors would like to thank the reviewers for their comments and suggestions, which have helped to improve this paper significantly.

References

1. W. M. P. van der Aalst. *Interval Timed Coloured Petri Nets and their Analysis*. Lecture Notes in Computer Science, vol. 691, pp. 451-472. 1993. 415
2. W. M. P. van der Aalst and M. A. Odijk. *Analysis of Railway Stations by Means of Interval Timed Coloured Petri Nets*. Real-Time Systems, vol. 9, pp. 241-263. 1995. 415
3. Parosh A. Abdulla and Aletta Nylén. *Timed Petri Nets and BQOs*. Proc. 22nd International Conference on Theory and Application of Petri Nets. Lecture Notes in Computer Science, vol. 2075, pp. 53-70. 2001. 415, 419
4. M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte and A. Cumani. *On Petri Nets with Stochastic Timing*. Proc. of the International Workshop on Timed Petri Nets, IEEE Computer Society Press, pp. 80-87. 1985. 415

5. E. Best, R. Devillers, J. Hall. *The Petri Box Calculus: A New Causal Algebra with Multi-label Communication*. Advances in Petri Nets, 1992. Lecture Notes in Computer Science, vol. 609, pp. 21-69. 1992. 414
6. T. Bolognesi, F. Lucidi and S. Trigila. *From Timed Petri Nets to Timed LOTOS*. Proceedings of the Tenth International IFIP WG6.1 Symposium on Protocol Specification, Testing and Verification. North-Holland, 1990. 415
7. Fred D. J. Bowden. *Modelling time in Petri nets*. Proc. Second Australia-Japan Workshop on Stochastic Models. 1996. 415
8. Antonio Cerone and Andrea Maggiolo-Schettini. *Time-based expressivity of time Petri nets for system specification*. Theoretical Computer Science (216)1-2, pp. 1-53. 1999. 415
9. D. de Frutos, V. Valero and O. Marroquín. *Decidability of Properties of Timed-Arc Petri Nets*. Proc. ICATPN 2000, Lecture Notes in Computer Science, vol. 1825, pp. 187-206. 2000. 419
10. U. Goltz. *On representing CCS programs by finite Petri nets*. MFCS, Lecture Notes in Computer Science, vol. 324 (1988), 339-350. 414
11. Hans-Michael Hanisch. *Analysis of Place/Transition Nets with Timed-Arcs and its Application to Batch Process Control*. Application and Theory of Petri Nets, LNCS vol. 691, pp:282-299. 1993. 415
12. P. Merlin. *A Study of the Recoverability of Communication Protocols*. PhD. Thesis, Univ. of California. 1974. 415
13. F. Moller and C. Tofts. *A Temporal Calculus of Communicating Systems*. CONCUR'90. Lecture Notes in Computer Science, vol. 458, pp. 401-415. 1990. 415, 419
14. J. Pardo, V. Valero, F. Cuartero and D. Cazorla. *Automatic Translation of TPAL specifications into Dynamic State Graphs*. To appear in the proc. of APSEC'01, IEEE Computer Society Press. 2001. 415, 416
15. C. Ramchandani. *Performance Evaluation of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD. Thesis, Massachusetts Institute of Technology, Cambridge. 1973. 415
16. G. M. Reed and A. W. Roscoe. *Metric Spaces as Models for Real-Time Concurrency*. Mathematical Foundations of Programming. Lecture Notes in Computer Science, vol. 298, pp. 331-343. 1987. 415, 419
17. J. Sifakis. *Use of Petri Nets for Performance Evaluation*. Proc. of the Third International Symposium IFIP W. G.7.3., Measuring, Modelling and Evaluating Computer Systems. Elsevier Science Publishers, pp. 75-93. 1977. 415
18. Dirk Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*. Lecture Notes in Computer Science, vol. 369. 1989. 414
19. V. Valero, D. de Frutos and F. Cuartero. *On Non-decidability of Reachability for Timed-Arc Petri Nets*. Proc. 8th Int. Workshop on Petri Nets and Performance Models, PNPM'99, pp. 188-196. 1999. 415, 419
20. V. Valero, D. de Frutos, and F. Cuartero. *Decidability of the Strict Reachability Problem for TPN's with Rational and Real Durations*. Proc. 5th. International Workshop on Petri Nets and Performance Models, pp. 56-65. 1993. 415
21. B. Walter. *Timed Petri-Nets for Modelling and Analysing Protocols with Real-Time Characteristics*. Proc. 3rd IFIP Workshop on Protocol Specification, Testing and Verification, North-Holland. 1983. 415
22. Wang Yi. *A Calculus of Real Time Systems*. PhD. Thesis, Chalmers University of Technology. 1991. 415, 419

Maria: Modular Reachability Analyser for Algebraic System Nets

Marko Mäkelä*

Helsinki University of Technology, Laboratory for Theoretical Computer Science
P.O.Box 9700, 02015 HUT, Finland
marko.makela@hut.fi
http://www.tcs.hut.fi/Personnel/marko.html

Abstract. Maria performs simulation, exhaustive reachability analysis and on-the-fly LTL model checking of high-level Petri nets with fairness constraints. The algebra contains powerful built-in data types and operations. Models can be exported to low-level Petri nets and labelled transition systems. Translator programs allow Maria to analyse transition systems as well as distributed computer programs written in procedural or object-oriented languages, or high-level specifications such as SDL. Maria has been implemented in portable C and C++, and it is freely available under the conditions of the GNU General Public License.

1 Introduction

1.1 Analysing High-Level Software Systems

There are many tools for analysing concurrent systems, but most of them are only suitable for education or for analysing relatively simple, hand-made highly abstracted models. At universities, many analysers have been developed just to see whether a theoretical idea might work in practice, often analysing models that do not directly have any roots in the real world. Commercial tool vendors concentrate on executable code generation and on graphical user interfaces.

Verifying industrial-size designs with minimal manual effort is a challenge. There may be no universal solution, but it is possible to list some requirements for automated checking of distributed software systems.

High-level formalism. The formalism used by the reachability analyser or model checker should have enough expressive power, so that high-level system descriptions can be modelled in a straightforward way, without introducing any superfluous intermediate states caused by having to translate, e.g., message buffer operations to non-atomic sequences of simpler operations.

* This research was financed by the Helsinki Graduate School on Computer Science and Engineering, the National Technology Agency of Finland (TEKES), the Nokia Corporation, Elisa Communications, the Finnish Rail Administration, EKE Electronics and Genera, and by a personal grant from Tekniikan Edistämissäätiö.

J. Esparza and C. Lakos (Eds.): ICATPN 2002, LNCS 2360, pp. 434–444, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Ease of use. Users should not need to be familiar with the formalism internally used by the analyser. The user works in the domain he is used to, and a *language-specific front-end* is responsible for hiding the underlying formalism:

- translate models to the internal formalism, abstracting from details that are not necessary in analysis
- allow desired properties to be specified in the design domain
- display erroneous behaviour in the design domain

Efficient utilisation of computing resources. The tools used in the modelling and verification process should be constructed in such a way that they work in a variety of computer systems, ranging from personal computers to multiprocessor supercomputers. The tools should not depend on the processor word length or byte order, and they should be based on standardised interfaces, such as [28]. Memory management should be optimised for large numbers of states and events.

1.2 Representing State Spaces

Efficient algorithms for exhaustive state space enumeration need to determine whether a state has been encountered earlier in the analysis. There are three fundamentally different approaches for representing the set of covered states.

Symbolic Techniques represent the set with a dynamic data structure that may be changed substantially when a state is added.
Explicit Techniques encode each state separately as a bit string.
Lossy Techniques transform states to hash signatures. In the event of a hash collision, substantial parts of the state space may remain unexplored.

It may be difficult to combine symbolic techniques with efficient model checking of liveness properties. Also, symbolic techniques have usually been implemented for formalisms having relatively simple data types and operations.

Since Maria supports a very high-level formalism, including, among others, operations on bounded queues, no efforts were made to implement symbolic techniques. When the state space management is based on explicit or lossy techniques, new operations on existing data types can be implemented independently.

1.3 Background

Maria [20] is the main product of a four-year research project that was carried out during the years 1998–2001. One of the most important goals in the project is to be able to directly analyse telecommunications protocols specified in SDL [29], the CCITT Specification and Description Language. In fact, the experience from Prod [26] and Emma [8] motivated the start of the whole project.

The goal was to develop a Petri net based state space exploration tool whose inscription language facilitates straightforward translation of data types and constructs found in SDL and high-level programming languages. Despite its expressive power, the Maria modelling language has a sound theoretical foundation. The semantics has been defined in [17] in terms of Algebraic System Nets [10].

The first usable versions of the analyser were released in the summer of 1999. Since then, Maria has been in extensive internal use, which has helped in finding and correcting errors and performance bottlenecks. In 2001, a graphical user interface for exploring state spaces and displaying query results was implemented on top of GraphViz [3]. In the fall of 2001, Maria replaced Prod as the main analysis tool in the education at our laboratory. On November 1, 2001, the tool was officially released as version 1.0.

The word "modular" in Maria refers to the software design of the tool, which makes it easy to incorporate different algorithms, front-ends and state storage mechanisms. The work on refinements [14] demonstrates that Maria can be used as a test-bed for new algorithmic ideas.

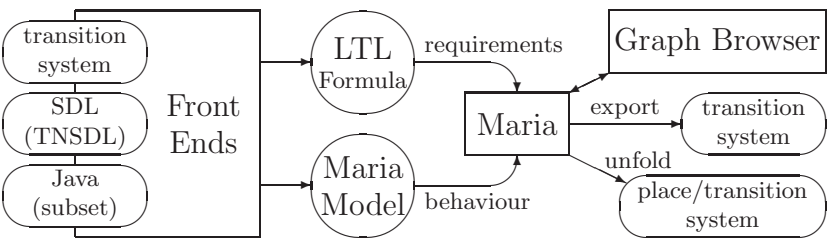


Fig. 1. The interfaces of Maria

2 Using Maria

Figure 1 illustrates the high-level interfaces of Maria. Models can be either written by hand or translated from other formalisms. A translator that allows Maria to model check parallel compositions of TVT [25] labelled transition systems is available from the home page [20]. Translators from SDL and Java are under development. The SDL translator [23] has been written from scratch, while one Java translator is based on Bandera [2].

Maria accepts commands from files, from a command line and from a graphical interface. Several modes of operation are supported:

- exhaustive reachability analysis with on-the-fly checking of safety properties
- interactive simulation: generate successors for states selected by the user
- interactive reachability graph exploration
- on-the-fly verification of liveness properties with fairness assumptions
- unfolding the model, optionally using a "coverable marking" algorithm [18]

The unfolding algorithms can output nets in the native input formats of PEP [7] and LoLA [22], as well as in the native format of Prod.

Figure 2 shows a simplified version of the distributed data base manager system, originally presented by Genrich and Lautenbach in [5]. The number of

```
typedef id[3] db_t;
typedef struct {
  db_t first;
  db_t second;
} db_pair_t;

place waiting db_t;
place performing db_t;
place inactive db_t: db_t d: d;
place exclusion struct {}: {};
place sent db_pair_t;
place recv db_pair_t;
place ack db_pair_t;

trans receive
in  { place inactive: r;
      place sent: { s, r }; }
out { place performing: r;
      place recv: { s, r }; };

trans ack
in  { place performing: r;
      place recv: { s, r }; }
out { place inactive: r;
      place ack: { s, r }; };

trans collect
in  { place waiting: s;
      place ack: db_t t (t != s): { s, t }; }
out { place inactive: s; place exclusion: {}; };

trans update
in  { place inactive: s; place exclusion: {}; }
out { place waiting: s;
      place sent: db_t t (t != s): { s, t }; };
```

Fig. 2. Distributed data base managers in the MARIA language. The number of data base manager nodes can be configured by modifying the first line

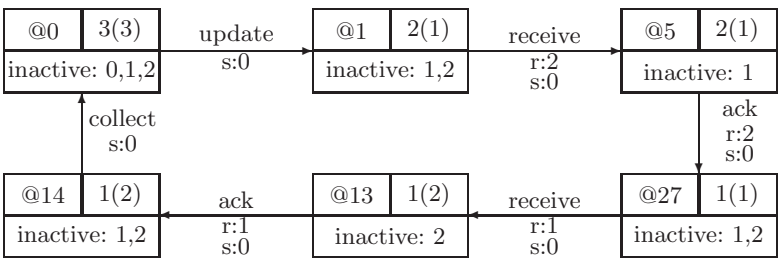


Fig. 3. An error trace for the LTL formula `<>place inactive equals empty` in the system of Figure 2. In this infinite execution, always at least one node is inactive

modelled nodes can be changed by altering the first data type definition; MARIA allows aggregate inscriptions in the transitions `collect` and `update`.

One use of MARIA is to verify liveness properties, such as the (incorrect) claim that in all executions starting from the initial state, all data base nodes are simultaneously active at some point of time. Error traces can be simplified by telling MARIA to only show the markings of certain places. The error trace in Figure 3 hides everything except the place named `inactive`.

3 Advanced Features

3.1 Powerful Algebraic Operations

The built-in algebraic operations in MARIA were designed to have enough expressive power for modelling high-level programs. In addition to the basic constructs familiar from programming languages such as C, there are operations for:

- managing items in bounded buffers (queues and stacks)
- basic multi-set operations (union, intersection, difference, mappings)
- aggregation: multi-set sums, existential and universal quantification

Aggregation over a dynamic range of indexes is a particularly powerful construct, because it allows arc expressions to be highly parameterisable. In Figure 2, the transition `update` models a broadcast operation with multi-set summation. For instance, when the variable `s` equals `3`, the formula `db_t t (t!=s): {s,t}` expands to the multi-set `{s,1},{s,2}`, or `{3,1},{3,2}`. If the first line of Figure 2 is modified to model n data base nodes, the multi-set will have $n - 1$ elements.

Basic algebraic operations check for exceptional conditions, such as integer arithmetic errors, queue or stack overflows or underflows, and constraint violations. The data type system is very sophisticated, and it is possible to arbitrarily restrict the set of allowed values even for structured types.

3.2 Optional C Code Generation

Transforming MARIA models to executable libraries is a nontrivial task, because operations on multi-sets, queues, stacks and tagged unions have no direct counterparts in the C programming language. In addition, all code must be instrumented with guards against evaluation errors and constraint violations.

The use of the compilation option [16] can speed up all operations except unfolding. Interpreter-based operation is useful in interactive simulations or when debugging a model, because the overhead of invoking a compiler is avoided.

3.3 Unifying Transition Instances and Markings

The unification algorithm that MARIA uses for determining the assignments under which transitions are enabled has been documented in [18]. In MARIA,

this depth-first algorithm has been implemented in such a way that enabled transition instances are fired as soon as they are found.

The interpreter-based implementations of the combined transition enabling check and firing algorithm use a search stack, while the compiler-based variant integrates the search stack in the program structure as nested loops.

3.4 Efficient State Space Management

Maria represents the states of its models in two different ways. The expanded representation is used when determining successor states and performing computations. Long-term storage of explored states is based on a condensed representation, a compact bit string whose encoding has been documented in [15].

By default, Maria manages reachability graphs (reachable states and events) in disk files. Keeping all data structures on disk has some advantages:

- the analysis can be interrupted and continued later
- the generated reachability graph can be explored on a different computer
- memory capacity is not a limit: a high-level model with 15,866,988 states and 61,156,129 events was analysed in 5 MB of RAM (and 1.55 GB of disk)

File system access can be notably slow even on systems that have enough memory to buffer all files. In some cases, more than half of the execution time of the analyser can be spent in determining whether a state has been visited. Enabling an option for memory-mapped file access reduces the analysis times of certain models to a sixth of the original. Unfortunately, with this option, the 4 GB address space of 32-bit systems is becoming a barrier. Really complex models can be analysed only with traditional file access, or with a 64-bit processor.

Optionally, Maria approximates the set of reachable states with a memory-based hash table. This option can be useful in cursory analysis of deadlocks and safety properties and for obtaining lower bounds for state space sizes.

3.5 Model Checking with Fairness Constraints

The on-the-fly LTL model checker in Maria takes into account both weak and strong fairness on the algorithmic level instead of adding it to the LTL specification. When a model has many fairness constraints, this can lead to exponential savings in time and space. To this end, the net class of the analyser includes constructs to flexibly express fairness constraints on transitions. To our knowledge, this is the first model checker of its kind for high-level Petri nets.

The model checking procedure [12,13] proceeds in an on-the-fly manner processing one strongly connected component of the product state space at a time. By detecting the presence of strong fairness constraints, the procedure tries to use generalised Büchi automata when possible to avoid the more complex task of checking the emptiness of Streett automata.

Generating short counterexamples when both weak and strong fairness are present can be quite challenging. Experiments with the algorithm that Maria uses have been reported in [12].

For translating temporal properties to generalised Büchi automata MARIA invokes an external tool that is a highly optimised implementation of the algorithm presented in [4]. Since the translator works as a textual filter, it can be easily replaced with an implementation of another algorithm.

4 Performance and Applications

4.1 Distributed Dynamic Channel Allocation

One of the first systems that was analysed with MARIA is a model of a radio channel allocation algorithm [21].

The sophisticated data type system and the aggregation operations of the modelling language allowed to write the model in a very compact way, and many modelling errors were quickly found in interactive simulation runs.

4.2 Verbatim Modelling of a Large SDL Specification

An important goal of MARIA was to be able to handle models of industrial-size distributed systems. The most complex system that has been analysed with MARIA so far is the *complete* radio link control (RLC) protocol of the third-generation mobile telephone system UMTS. The ETSI standard [27] describes this protocol in English, and the description is accompanied with 74 pages of informative graphical SDL diagrams that contain some inaccuracies and errors.

Each SDL statement was mechanically translated into a high-level net transition, generating hundreds of transitions. As one of the main functions of the protocol is to disassemble and reassemble data packets, the transmitted messages had to be modelled in detail. For one version of the model, MARIA encodes each reachable marking of the 142 high-level places in 167–197 bytes.

Timers were modelled in an abstract way, because the formalism of MARIA cannot describe time, but only the order in which events occur. Each timer was translated to a Boolean flag that indicates whether the timer is active. When a timer is activated, the flag is set. Resetting the timer clears the flag. Whenever the flag is set, a timeout can occur.

The parameters of the protocol model include message queue lengths, the domains of sequence numbers, and the type of communication channels. Both a reliable and a lossy channel have been analysed, but so far, the protocol has not been analysed on a channel that would duplicate or reorder messages. With the analysed initial parameters, the model has up to tens of millions of reachable states. LTL model checking has been applied on configurations that have less than 100,000 reachable states. The results will be reported in [24].

4.3 Benchmark: Distributed Data Base Managers

Although MARIA has a higher-level modelling language than PROD, using it does not imply a noticeable performance penalty. In the contrary, MARIA usually uses less memory (or disk) than PROD, and sometimes it consumes less processor time.

We have translated the PROD model of 10 distributed data base servers (sample file `dbm.net`) to MARIA format (file `dbm.pn` in the MARIA distribution). Figure 2 is a simplification of this model; among other things, it excludes capacity constraints and invariants that allow more compact representation of markings.

When PROD was invoked with a reasonably large `-b` parameter, it generated the 196,831 states and 1,181,000 arcs of the 10-server model in 262 seconds of user time and 10 seconds of system time. The 700 MHz Pentium III system had enough available memory to buffer the 69 megabytes of generated graph files.

We analysed the equivalent model in MARIA with the compilation option enabled. The analysis produced 22 megabytes of graph files in 130 seconds of user time and less than 1 second of system time. The improvement can be attributed to better state space encoding and to utilising a memory-mapped interface [28, Section 2.8.3.2] for accessing graph files.

5 Availability

MARIA is freely available as source code from the home page [20] under the conditions of the GNU General Public License. No registration is required.

It should be possible to compile the analyser for any environment that fully supports the C and C++ programming languages. Some features are, however, only present on Unix-like systems. The tool has been tested on GNU/Linux, Sun Solaris, Digital UNIX, SGI IRIX, Apple Mac OS X and Microsoft Windows.

6 Conclusion and Future Work

MARIA helps the analysis of industrial-size systems in multiple ways:

- by providing a theoretically sound modelling language that is suitable for describing high-level software systems,
- as an interactive simulator and visualiser of distributed systems, and
- as a "model checking back-end" for various formalisms ranging from labelled transition systems to SDL

Currently, MARIA does not support any state space reduction methods. Some algorithms on symmetry reduction [9] are being implemented. It has been planned to adapt partial order reduction methods [6] to MARIA.

The model checker in MARIA could treat safety properties in a more efficient way. Instead of interpreting the reachability graph, the property and their product as Büchi or Streett automata, it could interpret them as finite automata and thus avoid the costly loop checks. A translation from LTL to finite automata has been given in [11]. Recent developments in this area are documented in [19].

The present version of MARIA displays counterexamples as directed graphs whose nodes are reachable states in the model and edges are transition instances leading from a state to another. In many applications, it would be more intuitive to illustrate executions with message sequence charts [30]. In order for this to work nicely, a mechanism for specifying mappings from transition instances to messages and from markings to MSC conditions must be implemented.

Acknowledgements

The LTL model checker algorithm in MARIA was originally implemented by Timo Latvala. The design of MARIA has been influenced by feedback from Nisse Husberg, Teemu Tynjälä, Kimmo Varpaaniemi and others. The author would also like to express his thanks to the anonymous referees for their comments.

References

1. José-Manuel Colom and Maciej Koutny, editors, *Application and Theory of Petri Nets 2001, 22nd International Conference, ICATPN 2001*, volume 2075 of *Lecture Notes in Computer Science*, Newcastle upon Tyne, England, June 2001. Springer-Verlag. 443
2. James. C. Corbett, Matthew. B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri and Alexander Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM Press, New York, NY, USA. 436
3. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, September 2000. 436
4. Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland. 440
5. Hartmann J. Genrich and Kurt Lautenbach. The analysis of distributed systems by means of Predicate/Transition-Nets. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 123–146, Evian, France, July 1979. Springer-Verlag, 1979. 436
6. Patrice Godefroid, Doron Peled and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, July 1996. 441
7. Bernd Grahlmann. The state of PEP. In Armando M. Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST'98, Amazonia, Brazil*, volume 1548 of *Lecture Notes in Computer Science*, pages 522–526, Manaus, Brazil, January 1999. Springer-Verlag. 436
8. Nisse Husberg and Tapio Manner. Emma: Developing an industrial reachability analyser for SDL. In *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 642–661, Toulouse, France, September 1999. Springer-Verlag. 435
9. Tommi Junttila. Finding symmetries of algebraic system nets. *Fundamenta Informaticae*, 37(3):269–289, February 1999. 441
10. Ekkart Kindler and Hagen Völzer. Flexibility in algebraic nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN'98*, volume 1420 of *Lecture Notes in Computer Science*, pages 345–364, Lisbon, Portugal, June 1998. Springer-Verlag. 435

11. Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification: 11th International Conference, CAV'99*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183, Trento, Italy, July 1999. Springer-Verlag. 441
12. Timo Latvala and Keijo Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000. 439
13. Timo Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In [1], pages 242–262. 439
14. Glenn Lewis and Charles Lakos. Incremental state space construction for coloured Petri nets. In [1], pages 263–282. 436
15. Marko Mäkelä. Condensed storage of multi-set sequences. In *Workshop on the Practical Use of High-Level Petri Nets*, Århus, Denmark, June 2000. 439
16. Marko Mäkelä. Applying compiler techniques to reachability analysis of high-level models. In Hans-Dieter Burkhard, Ludwik Czaja, Andrzej Skowron and Mario Lenz, editors, *Workshop Concurrency, Specification & Programming 2000*, Informatik-Bericht 140, pages 129–141. Humboldt-Universität zu Berlin, Germany, October 2000. 438
17. Marko Mäkelä. A reachability analyser for algebraic system nets. Research report A69, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, June 2001. 435
18. Marko Mäkelä. Optimising enabling tests and unfoldings of algebraic system nets. In [1], pages 283–302. 436, 438
19. Marko Mäkelä. Efficiently verifying safety properties with idle office computers. Unpublished manuscript. 441
20. Marko Mäkelä. *Maria*. On-line documentation, http://www.tcs.hut.fi/maria/. 435, 436, 441
21. Leo Ojala, Nisse Husberg and Teemu Tynjälä. Modelling and analysing a distributed dynamic channel allocation algorithm for mobile computing using high-level net methods. *International Journal on Software Tools for Technology Transfer*, 3(4):382–393, 2001. 440
22. Karsten Schmidt. LoLA: A low level analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2001, 21st International Conference, ICATPN 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474, Århus, Denmark, June 2000. Springer-Verlag. 436
23. André Schulz and Teemu Tynjälä. Translation rules from standard SDL to Maria input language. In Nisse Husberg, Tomi Janhunen and Ilkka Niemelä, editors, *Leksa Notes in Computer Science: Festschrift in Honour of Professor Leo Ojala*, Research Report 63, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, October 2000. 436
24. Teemu Tynjälä, Sari Leppänen and Vesa Luukkala. Verifying reliable data transmission over UMTS radio interface with high level Petri nets. Unpublished manuscript. 440
25. Antti Valmari et al. Tampere Verification Tool. http://www.cs.tut.fi/ohj/VARG/. 436
26. Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen and Tino Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995. 435
27. *Universal Mobile Telecommunications System (UMTS); RLC protocol specification (3GPP TS 25.322 version 3.5.0 Release 1999)*. ETSI TS 125 322 V3.5.0 (2000-12). European Telecommunications Standards Institute, December 2000. 440

28. *Standard for Information Technology—Portable Operating System Interface*. IEEE Std 1003.1-2001. Institute of Electrical and Electronics Engineers, New York, NY, USA, December 2001. 435, 441
29. *CCITT Specification and Description Language (SDL)*. Recommendation Z.100. International Telecommunication Union, Geneva, Switzerland, October 1996. 435
30. *Message Sequence Chart (MSC)*. Recommendation Z.120. International Telecommunication Union, Geneva, Switzerland, November 1999. 441

Author Index

Aalst, Wil M. P. van der 1

Billington, Jonathan . 182, 273, 352
Buchs, Didier 142

Cheung, To-Yat 203
Cortadella, Jordi 80
Couvreur, Jean-Michel 101
Cuartero, Fernando 414

Dehnert, Juliane 121
Desel, Jörg 23

Encrenaz, Emmanuelle 101

Fleischhack, Hans 163

Glabbeek, Rob van 394
Gordon, Steven 182
Guelfi, Nicolas 142

Hayes, Ian J. 44

Jiao, Li 203

Kindler, Ekkart 217
Kondratyev, Alex 80
Kristensen, Lars Michael .. 182, 352

Lakos, Charles 59
Lautenbach, Kurt 237
Lavagno, Luciano 80
Le Ny, Louis-Marie 254
Liu, Lin 273
Lorentsen, Louise 294
Lu, Weiming 203

Mäkelä, Marko 434
Mailund, Thomas 314
Mandrioli, Dino 142
Marzo Serugendo, Giovanna Di 142

Nielsen, Mogens 335

Ouyang, Chun 352

Pardo, Juan José 414
Passerone, Claudio 80
Paviot-Adet, Emmanuel 101
Peuker, Sibylle 374
Poitrenaud, Denis 101

Stehno, Christian 163
Stork, David G. 394

Thiagarajan, P. S. 68, 335
Tuffin, Bruno 254
Tuovinen, Antti-Pekka 294

Valero, Valentín 414

Wacrenier, Pierre-André 101
Watanabe, Yosinori 80

Xu, Jianli 294

Yakovlev, Alex 70