Journal Subline
LNCS 7480

Wil M.P. van der Aalst · Gianfranco Balbo
Maciej Koutny · Karsten Wolf
Guest Editors

Transactions on Petri Nets and Other Models of Concurrency VII

Kurt Jensen
Editor-in-Chief

Springer

Lecture Notes in Computer Science 7480

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Kurt Jensen Wil M.P. van der Aalst
Gianfranco Balbo Maciej Koutny
Karsten Wolf (Eds.)

Transactions on Petri Nets and Other Models of Concurrency VII

Springer

ISSN 0302-9743 (LNCS) e-ISSN 1611-3349 (LNCS)
ISSN 1867-7193 (ToPNoC) e-ISSN 1867-7746 (ToPNoC)
ISBN 978-3-642-38142-3 e-ISBN 978-3-642-38143-0
DOI 10.1007/978-3-642-38143-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013937320

CR Subject Classification (1998): D.2, F.1, F.3, D.3, J.1, I.2.2

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

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface by Editor-in-Chief

The seventh issue of LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) contains material from the 5th International Summer School "Advanced Course on Petri Nets", held in September 2010 in Rostock, Germany. It was edited by Wil van der Aalst, Gianfranco Balbo, Maciej Koutny, and Karsten Wolf.

I would like to thank the four guest editors of this special issue: Wil van der Aalst, Gianfranco Balbo, Maciej Koutny, and Karsten Wolf. Moreover, I would like to thank all lecturers, reviewers, and the organizers of the advanced course, without whom this issue of ToPNoC would not have been possible.

February 2013

Kurt Jensen
Editor-in-Chief
LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)

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

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

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

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

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

ToPNoC contains:

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

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

Topics

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

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

For more information about ToPNoC, please see: www.springer.com/lncs/topnoc

Submission of Manuscripts

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

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

Preface

This volume contains material from the 5th International Summer School "Advanced Course on Petri Nets," held in September 2010 in Rostock, Germany. It followed the tradition of previous advanced courses, held in Hamburg (1978), Bad Honnef (1986), Schloss Dagstuhl (1996), and Eichstatt (2003). With a long and rich history as a stronghold of the Hanseatic League, almost 600 years of academic tradition, and as a touristic hotspot on the shores of the Baltic Sea, Rostock was a worthy host of the 5th gathering.

Due to their low frequency, advanced courses on Petri nets typically serve as a forum for important milestones in the area. In 2010, the program was compiled by the course director, Karsten Wolf, and the Scientific Board of the course, consisting of Wil van der Aalst (Eindhoven), Gianfranco Balbo (Turin), and Maciej Koutny (Newcastle). In the first week of the course, introductory lectures surveyed topics at the very core of Petri net research. In the second week, lectures covered areas that have received particular attention during recent years. This volume contains contributions from both weeks. Lecturers were invited based on their impact on the particular topic.

Nine lecturers accepted our invitation to transform their course material into a contribution to this volume. Naturally, the papers are more like surveys than regular research papers. Each paper was peer reviewed by a scientific board member and by a fellow author.

Deviating from the original structure of the course, this volume is grouped into three sections.

The first section is concerned with the creation of Petri net models and their validation. Van der Aalst et al. investigate the process of building complex Petri net models. The other two papers share rich experience in using Petri net models in particular domains. Kristensen and Simonsen discuss Petri net models for protocol designs while van Hee et al. consider Petri net models for business processes.

Papers in the second section address semantic issues and analysis methods. Best and Wimmel survey the Petri net structure theory, a research area with a long history. Koutny and Kleijn discuss issues arising in causality-based semantics for certain extensions to Petri nets. Valmari studies systems of state machines with variables, linking Petri nets to other formalisms.

The third section of this volume is devoted to the automatic synthesis of Petri nets. Reisig surveys the basic concepts of Petri net synthesis from a given transition system. Lorenz considers the generation of models from scenarios. Van der Aalst and van Dongen introduce the rapidly evolving area of process mining.

Only a few weeks before the advanced course, Carl Adam Petri passed away. This was intensely felt by lecturers and participants. In order to mark the

occasion, we invited G. Rozenberg, P.S. Thiagarajan, and W. Reisig to prepare a text in memoriam Carl Adam Petri.

We would like to thank the participants of the 5th Advanced Course on Petri Nets for the excellent atmosphere in the lectures and during the social events, and all lecturers for having delivered high-quality presentations. We are grateful to the authors and reviewers of this volume for meeting high scientific standards. The scientific board did a splendid job in compiling an interesting program, and the local team succeeded in running things smoothly.

The 5th Advanced Course on Petri Nets was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant WO 1466/14-1 and by Rostock University. Compilation of this volume was supported by the EasyChair system.

January 2013 Karsten Wolf

Program Committee

Gianfranco Balbo	University of Turin, Italy
Kurt Jensen	Aarhus University, Denmark
Maciej Koutny	Newcastle University, UK
Wil Van Der Aalst	Eindhoven University of Technology, The Netherlands
Karsten Wolf	University of Rostock, Germany

Additional Reviewers

Eike Best
Lawrence Cabac
Jetty Kleijn
Lars Kristensen
Robert Lorenz
Wolfgang Reisig
Natalia Sidorova
AnttiValmari

Table of Contents

In Memoriam: Carl Adam Petri

Wolfgang Reisig[1], Grzegorz Rozenberg[2,3], and P.S. Thiagarajan[4]

[1] Department of Computer Science, Humboldt Universität zu Berlin, Germany
[2] Leiden Institute of Advanced Computer Science,
Leiden University, The Netherlands
[3] Department of Computer Science, University of Colorado at Boulder, USA
[4] School of Computing, National University of Singapore

Carl Adam Petri was a visionary who founded an extraordinarily fruitful domain of study in the field of distributed discrete event systems. He was the first computer scientist to identify concurrency as a fundamental aspect of computing. He did so in his seminal PhD thesis from 1962 where in fact he outlined a whole new foundations for computer science. He devoted the rest of his working life to pursuing his ambitious and far reaching research goals. Petri nets -the core model that arose out of his thesis- have established themselves as a central model of distributed systems. They possess a rich theory, have been extended along multiple dimensions and are used in an astonishingly wide variety of domains.

Carl Adam passed away on July 2, 2010. His loss is felt deeply by friends and colleagues around the world. As a tribute, this contribution briefly surveys C. A. Petri's scientific life and impact.

1 The Early Years

Carl Adam Petri was born in Leipzig in 1926. He completed his Abitur in 1944 at the famous Thomasschule and was immediately drafted by the military. He was fortunate to be taken a prisoner of war by the British and remained in England until 1949. He then studied mathematics in Hannover and followed his teacher Heinz Unger to Bonn University as a PhD student. After receiving his PhD in 1962, he formed and managed the computer center of Bonn University. He then founded the Institute for Information Systems Research at the "Gesellschaft für Mathematik und Datenverarbeitung" (GMD) in Birlinghoven (currently a member institution of the Fraunhofer society). He directed this institute until 1991. In 1973 he seriously considered an offer of a Full Professorship at the University of Dortmund but in the end decided to decline.

Petri's early years had a strong influence on his later choice of scientific problems to focus on and on his very individualistic approaches to studying these

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 1–5, 2013.

problems. Petri's father was a serious scholar. He had a PhD in mathematics and had met Minkowski and Hilbert. He supported his son's interest in science. From a bookseller's bankrupt estate, Carl Adam received two thick textbooks on chemistry for his 12th birthday, which he worked through. His father also arranged for him special permission for the unrestricted use of the Leipzig central library where he delved into publications of Einstein and Heisenberg as a high school student. Later, as a flak auxiliary in the air force, he watched as his officers estimated the height, distance and speed of approaching aircraft by simple means including visual judgment and hearing. From this point on, the interplay between measurement and estimation and the inevitability of errors and their systematic treatment became a life long interest and influenced much of his scientific work.

2 Petri's Scientific Agenda

Carl Adam launched his scientific career with his dissertation "Kommunikation mit Automaten" ("Communication with Automata"), that he submitted to the Science Faculty of Darmstadt Technical University in July, 1961. He defended his thesis in June, 1962. [2].

This dissertation is a striking piece of work in multiple ways. From the opening lines it is clear one is dealing with an original and bold talent willing to challenge conventional wisdom. It lays out the case for a new theory of communication based on metric-free notions of time, space and causality. It argues for the necessity for such a theory in terms of reliably constructing and using information processing machines. Indeed from the very beginning, an implicit point of departure is that digital computers ought to be viewed as not number crunching numerical tools but as devices for storing, transforming and transmitting information. The ambiguous title of the dissertation which can be interpreted as communication *with* automata or *with the help of* automata also highlights this view ; and this was in 1962!

In order to cast the argument in a concrete setting Petri considers the problem of computing a recursive function using a physical device that obeys the known fundamental principles of physics. Since the amount of storage space needed for the computation can not be known in advance, if one starts with a finite amount of space then one will in general run out of space. However starting then the computation all over again with a larger amount of storage does not solve the problem due to the complexity of recursive functions. One will have to repeat this over and over again without making any progress. In the time domain there is an equally severe difficulty. Since only a bounded amount of information can be stored in a bounded volume (at the atomic level Heisenberg's uncertainty principle enforces this) in a synchronous architecture the clock pulses emanating from the system's central clock will have to travel longer and longer distances. Since there is an upper bound on the velocity of signals propagating through physical media (the special theory of relativity enforces a theoretical upper bound) this in turn will entail repeatedly reducing the clock speed. The inexorable conclusion

one reaches is that there can not be a synchronous computing system that can carry out universal computations in an effective way while being embedded in the known physical world.

Petri then offers an alternative approach based on organizing computations through strictly local and asynchronous operations where one part of the system can be extended as the need arises without disrupting the ongoing activities in the rest of the system. Specifically a potentially unbounded stack is constructed and since two such stacks can simulate the tape of a Turing machine one concludes that the proposed approach yields a device that can carry out universal computations while respecting fundamental physical principles.

We have described in some detail here just the technical core of this work. The dissertation as a whole sketches more or less the complete landscape of Carl Adam's research for the rest of his working life. In subsequent work he developed a number of formal notations for describing asynchronous distributed systems including graphical representations, algebraic formulae and topological constructs. He also coined the basic notions of Petri Nets, i.e. "places" and "transitions" to describe local states and actions, respectively to emphasize the core principle that states and changes of states must be represented and treated on an equal footing. Behavioral notions such as conflict, concurrency, causality, confusion and non-sequential processes also started to appear in his vocabulary.

Through the long years following his dissertation and nearly till the end Carl Adam pursued the grand vision laid out in his dissertation. He worked mainly alone but was always happy to explain the various parts of the theory he was trying to construct. This was done enthusiastically and at great length using beautiful examples while consuming endless cups of coffee and an unbroken chain of cigarettes. At the same time he was delighted to see the growth of net theory, especially its expanding application domains. He was an active and friendly presence in workshops, conferences and courses related to Petri nets and he was particularly keen to interact with students and young researchers.

3 The Evolution of the World of Petri Nets

The initial period of the Petri nets domain is somewhat hazy with independent strands of work pursued by different groups including Petri's group at GMD. To illustrate one such strand, the software pioneer Tom DeMarco came across Petri Nets at Bell Telephone Laboratories in the ESS-1 project that was developing the world's first commercial stored program telephone switch. DeMarco was a member of the project's simulation team. In his contribution to the volume on "Software Pioneers" [1], he writes "Among the documents describing the simulation was a giant diagram that Ms. Hoover (who led the team) called a Petri Net. It was the first time I had ever seen such a diagram. It portrayed the system being simulated as a network of sub-component nodes with information flows connecting the nodes. In a rather elegant trick, some of the more complicated nodes were themselves portrayed as Petri Nets....The one document that we found ourselves using most was Erna's Petri Net. It showed how all the pieces of

the puzzle were related and how they were obliged to interact. The lower-level network gave us a useful pigeon-holing scheme for information from the subsystem specs. When all the elemental requirements from the spec had been slotted by node, it was relatively easy to begin implementation. One of my colleagues, Jut Kodner, observed that the diagram was a better spec than the spec".

In a larger context, Anatol Holt played an influential role by recognizing the fundamental nature of Petri nets and bringing it to the attention of number of scientists in the US including Jack Dennis and his Computation Structures group at MIT. Suhas Patil, one of Dennis' PhD students saw the potential of Petri nets to specify and analyze asynchronous switching circuits and starting from his dissertation this became one of the early and very fruitful application domains for Petri nets. It was also soon recognized that Petri nets (more specifically, the version known as Place/Transition nets) constitute a very intriguing class of infinite state systems and a rich body of work regarding their relation to formal languages, decision problems and complexity classes began to be developed.

In Europe, the theory and applications of Petri nets grew at an increasing pace with many active groups establishing themselves in Denmark, France, Germany, Italy and Spain (to name a few). Equally important, the fact that concurrency was a fundamental aspect of computing was being recognized with Robin Milner's theory based on process algebras providing a major alternative impetus. Fundamental theoretical developments such as Mazurkiewicz trace theory and event structures as well as formal relationships between different models of concurrency started to appear. From the applications standpoint a crucial development was the formulation of related formalisms of Predicate/Transition nets and Colored Petri nets in which the tokens representing the local states as Boolean or integer-valued values were lifted in a coherent way to represent dynamic extensions of arbitrary multi-dimensional relations. This led to sustained research efforts resulting in a variety of system design tools accompanied by analysis and simulation methods.

As the applications of Petri nets grew so did the number of their variants with each domain demanding its own extension -often minor but sometimes major- of the basic formalism. Currently there are timed, stochastic, continuous and hybrid extensions of Petri nets that are well established. A large community of computer scientists and software engineers employ Petri Nets in a rich variety of settings. Petri nets are also deployed in other branches of engineering. As Prof. Gottzein in his laudatory speech (on the occasion of Carl Adam being awarded the 30th Werner-von-Siemens-Ring) put it: "Petri Nets brought engineers a breakthrough in their treatment of discretely controlled systems. Petri Nets are a key to solve the design problem, as this is the first technique to allow for a unique description, as well as powerful analysis of discrete control systems. Based on Petri Nets, it is now possible to formulate system invariants for discrete systems".

The field is active and constantly growing and what we have sketched here is a very brief, selective and incomplete account. It is perhaps too early to assess the full impact and influence of Carl Adam's contributions. This is particularly so since he has left behind a body of work that contains a wealth of

ideas and whose systematic development may well impact emerging alternative computational paradigms such as quantum computing. Independent of such future developments Petri's fundamental contribution to the theory and applications of distributed computing will endure. As Robin Milner in his Turing award lecture said: "Much of what I have been saying was already well understood in the sixties by Carl Adam Petri, who pioneered the scientific modeling of discrete concurrent systems. Petri's work has a secure place at the root of concurrency theory".

4 What Will the Future Bring?

In his invited speech at the 26th International Conference on Application and Theory of Petri Nets in Miami in June 2005, Petri appreciated the diversity and the quality of applications of his theory. But he called for a substantial expansion of the theory: Not in terms of additional Petri Net classes or more sophisticated analysis algorithms but rather for reaching the aims outlined in his dissertation. On this front, much remains to be explored! It may only be a matter of time until breakthroughs in hardware technologies coupled with vast demands of software will require the kind of net theory envisioned by Carl Adam. For instance conservation principles for information processing analogous to the conservation laws present in physics and chemistry may well be required in the future -as Petri speculated- to design and construct software interfaces connecting vast sources of dynamic data and applications that process such data.

The rapid rise of informatics has been driven mainly by technological advances and the economic imperative. Given its fundamental nature and its all encompassing influence, a rigorous *science* of informatics is undeniably needed. Such a science can evolve only by addressing the basic and far reaching questions raised by visionaries like Carl Adam Petri.

5 Conclusion

We wish to end on a personal note. Carl Adam was shy, modest and gentle. He had no guile or malice. He was a warm, gracious and entertaining host. He wore his exceptional scholarship lightly and it was a delight to be in his company. For those of us who got to know him well it was a priceless privilege.

References

1. Broy, D. (ed.): Software Pioneers. Springer (2002)
2. Petri, C.A.: Kommunikation mit Automaten. Schriften des Rheinisch-Westfälischen Instituts für Instrumentelle Mathematik 2. Universität Bonn (1962)

Strategies for Modeling Complex Processes Using Colored Petri Nets

Wil M.P. van der Aalst, Christian Stahl, and Michael Westergaard

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands
{W.M.P.v.d.Aalst,C.Stahl,M.Westergaard}@tue.nl

Abstract. Colored Petri Nets (CPNs) extend the classical Petri net formalism with data, time, and hierarchy. These extensions make it possible to model complex processes as CPNs without being forced to abstract from relevant aspects. Moreover, CPNs are supported by CPN Tools—a powerful toolset that supports the design and analysis of such processes. The expressiveness of the CPN language enables different modeling approaches. Typically, the same process can be modeled in numerous ways. As a result, inexperienced modelers may create CPNs that are unnecessarily convoluted and bulky. Using a running example and a set of design patterns, we show how to solve typical design problems in terms of CPNs. By following these guidelines, it is possible to create succinct, but also comprehensible, models. In addition, we present some new features supported by CPN Tools 3.0 (e.g., priorities and real time stamps) and show how the software can be used for performance analysis (i.e., comparing design alternatives using simulation).

Keywords: Colored Petri nets, Design patterns, CPN Tools.

1 Introduction

Petri nets have been around for about half a century and have shown to be able to model concurrent processes adequately. The basic formalism is simple and enables powerful analysis techniques. However, it is not easy to model complex processes in terms of classical Petri nets. Therefore, many extensions of the basic formalism have been proposed in the literature [1,13,14,15,17,18,19,20,22,23,30]. In fact, hundreds of extensions have been proposed for classical Petri nets, and it is impossible to name them all here. Some of the extensions proposed are rather exotic and did not progress beyond a proposal on paper (i.e., no tool support and no practical applications), whereas other extensions are widely supported and frequently used. Despite the many proposals, there seems to be consensus on the need for three types of *extensions*:

– *The extension with data.* In the classical Petri net, two tokens *cannot* be distinguished. The only way to distinguish two tokens is to put them in separate places. This is not practical for realistic applications as the model

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 6–55, 2013.

quickly becomes extremely complex and potentially infinitely large. In fact, for most practical applications of Petri nets, we would like the tokens to be distinguishable and have particular characteristics (e.g., age, weight, price, value, owner, or address). For a token modeling a car, we may want to describe its brand, model, color, or license number. Therefore, we need to add data to the basic model. Without this extension, Petri nets are like a programming language without variables and parameters.

- *The extension with hierarchy.* No matter how expressive a modeling language is, models tend to become large because in most applications there are many entities that interact in a nontrivial manner. Therefore, a hierarchy concept is needed to deal with this type of complexity. When designing a model one would like to use a divide-and-conquer approach. Moreover, structuring a model is essential when communicating design choices and analysis results with stakeholders. Without hierarchy, Petri nets are like a programming language lacking subroutines and subprocedures.
- *The extension with time.* Petri nets are often used to model processes where time plays an important role. For example, activities take time or exception handling is required after a timeout. These temporal aspects should be reflected in the model. Durations may be deterministic or stochastic. In the latter case, the model typically also incorporates routing probabilities such that performance analysis comes into reach. In many application domains it is important to use models to predict response times, utilization, flow times, and service levels.

Although there is consensus on the need to support data, hierarchy, and time for practical applications of Petri nets, different proposals have been made. Some of the differences between competing proposals are mainly syntactical; for example, CPN Tools is using ML as an inscription language [19,24] whereas ExSpect is using a dedicated functional language [3,15]. Other differences are more relevant; for example, the hierarchy concept used in CPN Tools (transition refinement) is very different from the nets-in-nets paradigm used by Renew [21,30].

Colored Petri Nets (CPNs) are the most widely used formalism incorporating data, hierarchy, and time [6,17,18,20,19]. Initially, CPNs were supported by *Design/CPN*. Later, Design/CPN was replaced by *CPN Tools*.[1] Currently, CPN Tools is by far the most widely used Petri net tool. CPN Tools supports the design of complex processes and the analysis of such processes using state-space analysis and simulation.

Modeling complex processes in terms of CPNs is a nontrivial task. The expressiveness of the language allows for different styles of modeling; that is, the same process can be modeled in different ways. Modeling is "an art rather than a science", but there are recurring modeling problems that can be solved by applying *design patterns*.

The most well-known patterns collection in the IT domain is the set of design patterns documented by Gamma, Helm, Johnson, and Vlissides [12].

[1] See `http://cpntools.org`

This collection describes a set of problems and solutions frequently encountered in object-oriented software design. The success of the patterns described in [12] triggered many patterns initiatives in the IT field, including the Workflow Patterns Initiative [4,32]. The idea to use a patterns-based approach originates from the work of the architect Christopher Alexander. In [7], he provided rules and diagrams describing methods for constructing buildings. The goal of the patterns documented by Alexander is to provide generic solutions for recurrent problems in architectural design. The idea to use patterns for design problems in the IT domain is appealing as is reflected by the different collections of patterns [4,5,10,12,16,29,31,32]. Many of these collections focus on behavioral aspects as these are most difficult to model and implement.

The idea to provide patterns for modeling in terms of CPNs was first proposed in [26]. Based on expert opinions and an analysis of large collections of CPNs (taken from papers and Web pages), 34 patterns were identified (see Appendix). These patterns help to tackle particular problems. Each pattern is described using a standard format including elements such as pattern name, intent, motivation, problem description, solution, implementation considerations, examples, and related patterns. In this paper, we explain the most important patterns using a running example. Unlike in [25,26], we do not explicitly enumerate the patterns nor will we use a strict format to present them. Instead, we use a tutorial-style presentation showing how to address frequently recurring modeling problems.

For a detailed description of the CPN language we refer to [18,19]. Our goal is not to describe the language but to focus on the way it can be used most efficiently. This paper is based on lectures given by the authors during the 5th Summer School/Advanced Course on Petri nets (Rostock, September 2010). Hence, the goal is not to present new scientific results, but to guide people using the CPN language and CPN Tools. In addition, the paper presents recent extensions of CPN Tools. As of CPN Tools 3.0, priorities and real time stamps are supported. We shall show that these extensions provide additional support when tackling some of the most important design patterns.

In the last part of the paper, we focus on one particular analysis technique: *simulation*. We shall show that the timing concept used by CPNs is compelling and gives the designer full control over temporal aspects of the model. Moreover, CPN Tools provides a powerful simulation environment. Using our running example, we shall show that it is easy to compare different alternative models.

The remainder of this paper is organized as follows. Section 2 introduces the basics of CPNs. The focus is on the extension with data, that is, colored tokens. As a running example, we use a gas station that serves two types of customers. The extension with hierarchy is described in Sect. 3. Subsequently, we use different variants of the gas station to explain four of the simple design patterns (Sect. 4) and two of the more advanced design patterns (Sect. 5). The hierarchy concept is used to structure these patterns while the patterns themselves focus on the interplay between control-flow and data. Here, we also show how the priority concept of CPN Tools 3.0 can be used to simplify the realization of some of the patterns. In Sect. 6, we shift our attention to modeling of time. We explain

the time concept and highlight the new timing functionality of CPN Tools 3.0 (i.e., real time). Subsequently, Sect. 7 shows how the addition of stochastic elements can be used to simulate complex processes and analyze their performance. Section 8 concludes the paper.

2 Colored Petri Nets: Basics

Colored Petri nets (CPNs) extend classical Petri nets with data, hierarchy, and time. In this section we focus on the *extension with data*.

In a classical Petri net, tokens are indistinguishable (black), whereas in CPNs tokens are distinguishable; tokens may have different colors such that they can be differentiated. A colored token carries data attributes that characterize the entity it represents. Note that we use the terms "color" and "data" interchangeably. This extension enables us to explicitly model concurrency using the power of Petri nets but also to model sequential or data-processing systems using a programming language, leading to much more compact and precise models, especially if several entities in a system behave in a similar manner. That way, CPNs provide a modeling technique that enables us to model complex systems in detail.

As a running example, we consider a gas station. We start with a simple version, and, throughout the paper, we add more features. This will illustrate how to construct a model by incrementally refining and extending it. Moreover, it allows us to show various design patterns for CPNs.

In our example, cars arrive at the gas station, wait to be served or rejected if the station is lacking capacity, and finally leave. We distinguish between regular cars and taxis. Figure 1 shows the first version of a CPN model. In the rest of this section, we explain this example in detail and use it to introduce the CPN formalism. We do not give a formal definition of CPNs, but mention that one such exists and can be found in, for instance, [18,19].

Like for classical Petri nets, the basic components of CPNs are *place*s, *transition*s, and *arc*s. A place serves as a placeholder for the entities in the system and

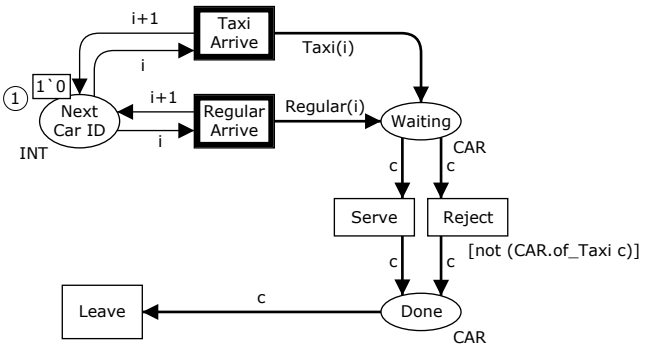


Fig. 1. Simple CPN model of a gas station

is represented by an ellipse. There are three places in Fig. 1: Next Car ID, Waiting, and Done. A transition represents an action of the system. Graphically, a rectangle represents a transition. The CPN in Fig. 1 has five transitions: Taxi Arrive, Regular Arrive, Serve, Reject, and Leave. Places and transitions are connected by directed arcs, which describe how data flows when transitions are executed, but we defer the exact description for a moment. An arc can only connect a place to a transition or a transition to a place. An arc between two places or between two transitions is not possible. So a CPN induces a bipartite directed graph with places and transitions as nodes.

Each place has a *type* (also known as a color set or sort) that determines which kind of tokens it may contain. This is comparable to how variables have a type in (explicitly) typed languages and is used both to make it easier to understand the model and to catch errors. In Fig. 1, place Next Car ID is of type INT and the places Waiting and Done are of type CAR. This indicates that we can only have integers in Next Car ID and only cars in the two remaining places. Types are declared explicitly in the model using the language CPN-ML, which extends Standard ML [24] with syntax for CPNs. In CPN-ML, the declarations of the types INT and CAR are:

```
colset INT = int;
colset CAR = union Regular: INT + Taxi: INT;
```

The first line indicates that the type INT corresponds to the simple type int (integer). Declarations allow us to give different names to simple types to make the model more readable (e.g., defining a type ID if we were using integers as identifiers). The second line specifies that CAR is a union type, which corresponds to a *datatype* in Standard ML or a disjoint union in mathematics. The idea is that we can have values that are either Regular cars or Taxis. Regular cars and taxis have an associated integer, which we use to be able to distinguish each individual car. Thus, the type CAR contains the values {Regular(0), Taxi(0), Regular(1), Taxi(1), ...}.

Aside from a type, each place also has a *marking*. A marking of a place is a multiset of values over the type of the place. A multiset is like an ordinary set (i.e., the order of elements does not matter), but the same element can occur multiple times. A token is an element of such a marking; that is, it has a value and resides in a place. In the example in Fig. 1, markings of places are shown in a circle and a rectangle near the places, such as the circle containing 1 and the rectangle containing 1‘0 on place Next Car ID. The number in the circle represents the total number of tokens in the place, and the text in the rectangle is a textual representation of the multiset of tokens. In the example, place Next Car ID contains exactly one token and the marking is written 1‘0. We use a backwards apostrophe (‘) to separate the value of a token and the count of how many tokens with that value is part of the marking. The marking in Fig. 1 consists of one token with value 0. If a marking consists of tokens with different values, we separate them with two pluses (++). This allows us to write a marking such as 2‘1++3‘5 to represent the multiset containing two tokens with value 1 and three tokens with value 5. Another example of a marking is 1‘"Hello"++1‘"World" for

a marking containing two tokens, one with value "Hello" (i.e., the string Hello) and one with value "World". Places without a marking shown contain an empty multiset which is not shown explicitly. An assignment of markings to all places is a *marking* of the net (or model).

We think of arcs as belonging to transitions, and separate them into *input arc*s and *output arc*s. An input arc connects a place to a transition, and an output arc connects a transition to a place. An arc has an inscription, i.e. an *expression*, written in CPN-ML. Expressions are like expressions in programming languages and may contain constants, functions, and all common arithmetic operators. An expression may contain one or more free *variables* (i.e., it may be an open expression). In Fig. 1, transition Regular Arrive has an input arc from place Next Car ID with expression i and two output arcs—one to Next Car ID with expression i+1 and one to Waiting with expression Regular(i). A place connected to a transition using an input arc is an *input place*, and a place connected using an output arc is an *output place*. In the example, Next Car ID is an input place of transition Regular Arrive, and Next Car ID and Waiting are output places of the same transition. A place can thus be an input and an output place of the same transition. Variables must be declared to be of a certain type. In our example, we have declared two variables:

```
var i: INT;
var c: CAR;
```

Variable i is of type INT and variable c of type CAR. An expression on an arc must have the same type as the type of the place it is connected to or a multiset of the place type; that is, when a value (of correct type) is assigned to all free variables in an expression, it must evaluate to a multiset over or a single value of the type of the place the arc is connected to.

A transition has a natural set of variables, namely the ones occurring on all arcs belonging to it. Each of these variables can be assigned a value from the set represented of its type. For example, the variable i can be assigned the value 0, 1, or 37 as they are all integers. We refer to a transition along with an assignment to each of its variables as a *binding element* (or binding for short). We denote a binding element by the name of the transition and a list of assignments to all its variables in braces. In our example, there are binding elements Regular Arrive⟨i=0⟩, Regular Arrive⟨i=37⟩, Serve⟨c=Taxi(23)⟩, and many others. Note that such potential bindings exist independent of a particular marking; that is, when talking about binding elements we do not look at surrounding places, but only consider the free variables of arcs surrounding the transition. We note that even though variable i occurs on more than one arc connected with Regular Arrive, we only write it once in a binding element. If the transition is clear from the context, we may omit the name of the transition when talking about a binding element.

Given a binding element, we can evaluate the expressions on all arcs belonging to the transition. For example, given the binding element Regular Arrive⟨i=0⟩ in Fig. 1, the expression i of the input arc from Next Car ID evaluates to 0, the expression i+1 on the output arc to Next Car ID evaluates to 0+1=1, and the

expression Regular(i) on the output arc to Waiting evaluates to Regular(0). For the binding element Regular Arrive⟨i=37⟩, the same expressions evaluate to 37, 38, and Regular(37), respectively. As we only write each variable once, it has to have the same value in all expressions surrounding a transition, but it can have other values on other transitions. That means, the scope of a variable in a CPN model is a transition, and information cannot be exchanged between transitions directly. We can think of a transition as inducing a namespace for all variables surrounding it.

Given a model with a marking and a binding, we say that the binding is *enabled* if all input places contain at least the tokens specified by the evaluation of the expression on the corresponding input arc in the binding. In Fig. 1, the binding element Regular Arrive⟨i=0⟩ is enabled as the expression on the sole input arc to the transition evaluates to 0, and the marking of Next Car ID contains a token with value 0. The binding element Regular Arrive⟨i=37⟩ is not enabled in the marking in Fig. 1 as Next Car ID does not contain a token with value 37.

A transition is *enabled* in a marking if there exists at least one binding element which is enabled in the marking. In Fig. 1, we have two enabled transitions, Taxi Arrive and Regular Arrive, as evidenced by the enabled binding elements Regular Arrive⟨i=0⟩ and Taxi Arrive⟨i=0⟩. Enabled transitions are marked using a bold outline. Figure 1 shows that in the initial marking both Regular Arrive and Taxi Arrive are enabled. Each of them has one enabled binding: Regular Arrive⟨i=0⟩ and Taxi Arrive⟨i=0⟩.

If a binding element (or a transition) is enabled, it can *occur* or be *executed*. This has the effect of removing all tokens from input places corresponding to evaluations of expressions on input arcs and producing new tokens on output places corresponding to evaluations of expressions on output arcs. In the example, if binding element Regular Arrive⟨i=0⟩ occurs, we get a situation like the one in Fig. 2. Compared to the situation in Fig. 1, only the marking has changed; the net structure remains unchanged. When Regular Arrive⟨i=0⟩ occurs, it consumes the token with value 0 from Next Car ID and produces a token with value 1 according to the arc expression i+1 in Next Car ID. In addition, it produces a token with value Regular(0) in Waiting, and this place now contains exactly one token with this value. Transitions Taxi Arrive and Regular Arrive remain enabled in this marking, albeit the enabled bindings changed: Taxi Arrive⟨i=1⟩ and Regular Arrive⟨i=1⟩. Furthermore, transitions Reject and Serve are now enabled, both in a binding ⟨c=Regular(0)⟩.

We look at the net structure of a model and its marking as separate things; the marking of the model may change, but the net structure remains the same. The marking of a model before we start simulation is the *initial marking*; for example, in Fig. 1, only place Next Car ID contains a token; this token has value 0. The marking after executing Regular Arrive⟨i=0⟩ is depicted in Fig. 2. We refer to such a marking as the *current marking*. Executing binding Taxi Arrive⟨i=1⟩ yields a new current marking shown in Fig. 3. Execution of a binding element is a *step*. Transitions Taxi Arrive and Regular Arrive, thus, intuitively model that

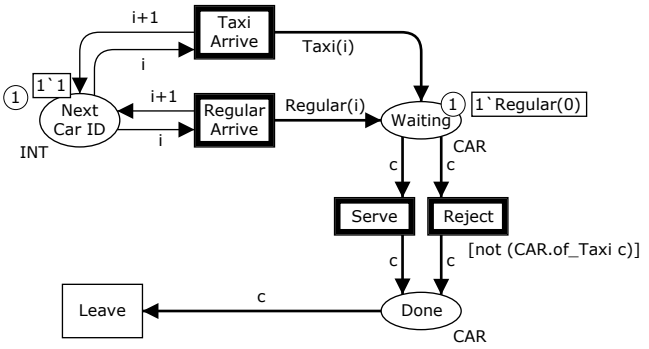


Fig. 2. Gas station model after executing binding element Regular Arrive⟨i=0⟩

a car of the given type arrives and queues up in Waiting. We use the token in Next Car ID as a counter to number all cars arriving.

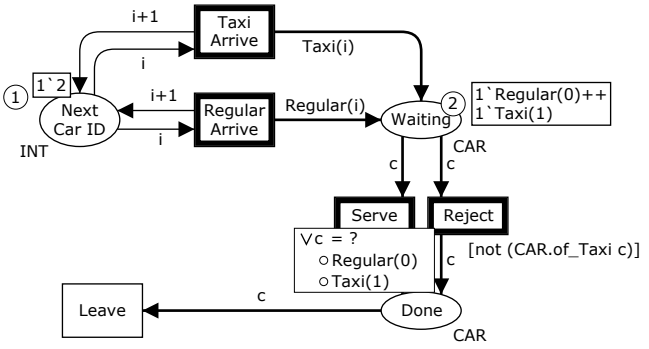


Fig. 3. Gas station model after Regular Arrive and Taxi Arrive have been executed. As shown, transition Serve is enabled in two bindings, ⟨c=Regular(0)⟩ and ⟨c=Taxi(1)⟩.

In Fig. 3, transition Serve is enabled in two bindings, ⟨c=Regular(0)⟩ and ⟨c=Taxi(1)⟩. These two bindings are shown in Fig. 3; the rectangle near the transition shows that c has two possible values enabling Serve.

If we execute the transition in the binding Serve⟨c=Taxi(1)⟩, we obtain the situation in Fig. 4, where the Taxi(1) is removed from Waiting and a token with the same value has been produced in Done. Intuitively, this models the serving of a taxi at the gas station and moving it to a place where it is no longer waiting to be served. Now, the taxi can leave (transition Leave is enabled), leading to a marking similar to Fig. 2 except the marking of Next Car ID is 1‘2 instead of 1‘1.

Figures 1, 2, 3, and 4 show one possible sequence of steps. Executing enabled bindings and thus moving from one marking to another is also known as the *token*

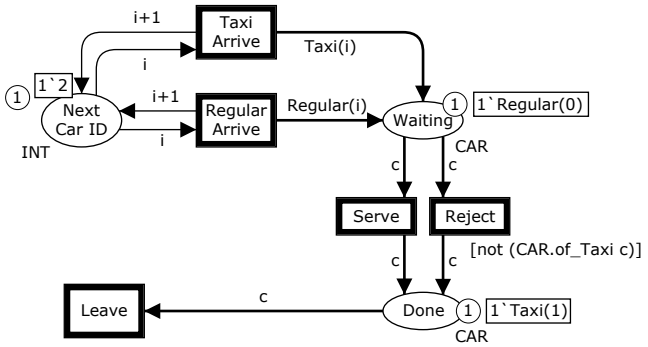


Fig. 4. Gas station model after serving the taxi

game. This is the mechanism used when simulating the CPN. Note that whenever a state has multiple enabled bindings, one needs to pick one of these bindings.

If we reconsider the marking in Fig. 3, we see that transition Reject has an inscription in squared brackets to the left of it, namely [not (CAR.of_Taxi c)]. This is a *guard*. A guard defines an additional constraint that must be fulfilled before a transition is enabled; that is, it is a Boolean expression that needs to evaluate to true in addition to the earlier requirements. When a guard is not shown, it implicitly always evaluates to true. A guard may also contain free variables, and they are considered in the same way as free variables on arcs surrounding a transition when considering bindings of the transition. In our example, the guard uses a function, CAR.of_Taxi, automatically defined for union types, which returns whether the parameter given is a Taxi (regardless of the integer value). Thus, the guard evaluates to true if the value of c is not Taxi. This models that we do not wish to reject taxis, for example, because they bring a lot of business. This semantics of a guard is also reflected in the enabled bindings of transitions as shown in Fig. 5, where the binding Reject⟨c=Regular(0)⟩ is enabled, but Reject⟨c=Taxi(1)⟩ is not, even though the token required is present.

3 Hierarchical Modeling

In this section, we present an approach to extend CPNs with *hierarchy*. This approach makes it possible to reflect the hierarchical structure of the system in the CPN model. Hierarchical CPNs simplify modeling, thus facilitating the modeling of large and complicated systems. The idea is to decompose a system into a set of *modules*. A module is a CPN with a set of interface places, and it can be used to describe the internal structure of a substitution transition. By showing the substitution transition at the higher level, we can abstract from the inner structure of a module at the lower level.

First, we present hierarchical CPNs, as supported by CPN Tools and show how hierarchical modeling can be used to refine our running example. Subsequent, we sketch different approaches of hierarchical modeling.

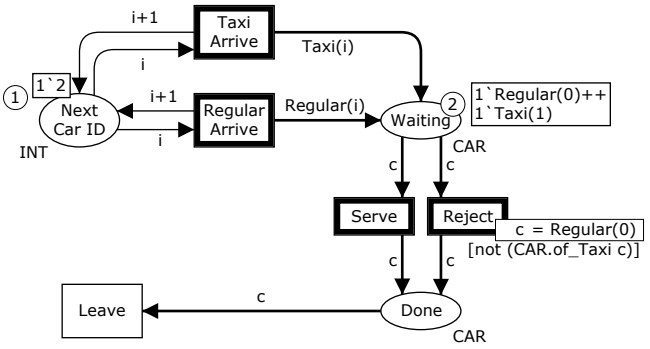


Fig. 5. Gas station model: Enabling of Reject is affected by its guard

3.1 Hierarchical CPNs

CPNs, as introduced in the previous section, are suitable to model the behavior of complex systems. The concept of place types allows us to specify the flow of data objects of any data type, and by using guards and arc inscriptions we can manipulate these data objects. The weakness of "flat" CPNs is that they try to capture the system behavior in one comprehensive net and do not represent the hierarchical structure of a system. For example, the CPN in Fig. 1 models the serving and rejecting of cars at the gas station but also how cars arrive and leave the gas station. In other words, the CPN in Fig. 1 is unstructured.

For toy examples like the CPN modeling a gas station, this is not a problem. If a CPN has only few places and transitions, we can lay out the net structure in a way such that the individual parts of the system can be recognized. However, if we model more complex systems such as a more refined version of the gas station example, then the resulting CPN model could have hundreds of places and transitions. Such models cannot be overseen and are, therefore, not suitable for discussing design decisions or implementation details of a system.

In a CPN, we model the elements of a system as places, transitions, and tokens. These elements do not allow us to structure a model. As a consequence, modeling a system by using only places, transitions, and tokens is insufficient. Concepts to abstract from parts of the model are necessary. To this end, models are usually designed following a *hierarchical approach*. The idea is to have several levels of abstraction of the system and to refine elements at higher levels into more detailed elements at lower levels. That way, also the design of large and complex systems becomes manageable, because designers usually concentrate on a single aspect of a system and extend the model step by step. For example, if we only want to know when a car is rejected at the gas station, then the information of arriving and leaving cars is not relevant and should be abstracted from. We illustrate the idea of hierarchical modeling by revisiting our running example introduced in Fig. 1.

We structure the model in Fig. 1 by decomposing it into two modules: (1) the gas station and (2) its environment. The gas station module represents the

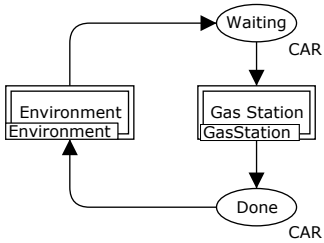


Fig. 6. CPN model of the gas station top module

functionality of the gas station—that is, the serving and rejecting of waiting cars, as modeled by transitions Service and Reject. The environment module models the arrival and leaving of the cars, as modeled by transitions Taxi Arrive, Regular Arrive, and Leave. The interface between these two modules can be specified by places Waiting and Done. We refer to such a place as a *socket*. Both socket places are of type Car. A token in Waiting models a car waiting to be refueled, and a token in Done models a car that has been refueled or rejected. Figure 6 shows the corresponding CPN model of this top-level module. The double-lined rectangles, which are labeled Environment and Gas Station, denote the two respective modules in an abstract manner.

As already mentioned, a module is a CPN consisting of places, transitions, arcs, and tokens. There are two kinds of transitions: *elementary transitions* and *substitution transitions*. An elementary transition is an ordinary transition, as introduced in the previous section. A substitution transition refers to a module. It abstracts from the internal behavior of a module; that is, it considers a module as a black box. Unlike a normal transition, a substitution transition may have internal states and does not need to consume and produce tokens in one atomic action. For example, depending on the underlying module, Gas Station may first consume ten cars from place Waiting, before it produces a token to Done.

A module may contain any number of substitution transitions. These substitution transitions refer to other modules that, in turn, may contain transitions referring to other modules. There can be an arbitrary many levels as long as no cycles are introduced in the inclusion graph; that is, a module may not (transitively) contain itself as this would correspond to an infinitely large model when we replace each substitution transition by the module it refers to.

The top-level module in Fig. 6 has two substitution transitions Gas Station and Environment referring to modules Environment and GasStation, respectively, as can be seen from the tag on the bottom of a substitution transition. Figure 7(left) shows the CPN modeling module Environment and Fig. 7(right) the CPN modeling module GasStation.

Each of the two CPNs in Fig. 7 has two interface places: Waiting and Done. We refer to such a place as a *port*. To be able to replace a substitution transition by the CPN modeling the module it refers to, we need to relate each socket of the substitution transition to a port of the CPN modeling the module. That way, pairs of places—a port and a socket—are semantically merged into one

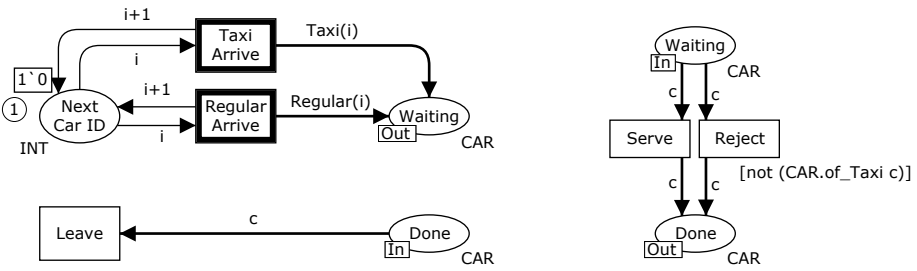


Fig. 7. CPN models of the environment (left) and gas station module (right)

place. As an example, sockets Waiting and Done in Fig. 6 are merged with the equally labeled ports in Fig. 7(right). On the level of a port, we specify the type of connection; that is, a port has a *type*. In Fig. 7(right), the tag In on place Waiting denotes that this port is of type input. Likewise, the tag Out on place Done shows that this port is of type output. A port can also be of type input and output. In this case, it is labeled I/O.

Replacing a substitution transition by the module it refers to is called the *flattening* of a CPN. The semantics of the CPN in Fig. 6 corresponds to the flat CPN in Fig. 1.

Another advantage of decomposing a system into modules is that it enables us to *reuse* existing functionality. That is, the same module can be used several times in a model if necessary. As a result, multiple substitution transitions may refer to the same module. To cope with this, we distinguish between a *module definition* and a *module instance*. Whereas a module definition serves as a specification of a CPN model, a module instance can be seen as an individual copy of the module definition. For example, we could extend our model in Fig. 6 and connect sockets Waiting and Done with an additional gas station, say Gas Station 1. In this case, we have two substitution transitions, Gas Station and Gas Station 1, both referring to the same module definition, GasStation. However, each substitution transition would be replaced by an individual module instance—that is, a separate copy of the CPN shown in Fig. 7(right).

Finally, we illustrate how hierarchical modeling simplifies the design and, in particular, the refinement of a system. Consider the module of the gas station, as depicted in Fig. 7(right). Cars are modeled as tokens in place Waiting and are waiting to be served. In the current model, a car may be waiting to be served and after a while be rejected. This is not desirable. Therefore, we extend the model as follows: the gas station has some waiting space where cars are queueing. If the gas station has too little capacity, arriving cars will be rejected. However, once a car enters the queue at the gas station, it will be eventually served.

To modify the model, we only need to refine the CPN modeling the gas station module (see Fig. 7(right)). Figure 8 shows the resulting model. An arriving car is either rejected right away or put in the queue. As in the previous models, taxis are not rejected. Place Queue models the queue at the gas station.

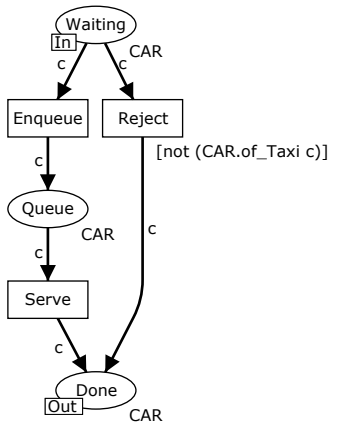


Fig. 8. Improved CPN model of the gas station module

The advantage of hierarchical modeling is that we need only to refine the module of the gas station. As the environment is not affected by the change, we do not have to touch the respective model. This example shows that by using hierarchical CPNs, designers can focus on single aspects of a model.

3.2 Approaches

There are two prominent approaches to obtain a hierarchical model for a system: the *top-down approach* and the *bottom-up approach*.

In the top-down approach, we start at the highest level of abstraction. and *decompose* the system into modules. Each module is considered as a black box, and only the relationship between the modules, which is modeled as an interface, is relevant. In subsequent steps, we can consider each module as a black box and refine it into a set of submodules. We can repeat this procedure until we have reached the desired degree of abstraction.

The bottom-up approach starts at the lowest level of abstraction and works in the opposite direction as the top-down approach. At this level of abstraction, we describe the elementary modules in detail. These modules are then *composed* to form a compound module. This composition step is repeated until we reach the highest level of abstraction at which the system is modeled as a single module.

Hierarchical development of systems is widely used, and all modern programming languages offer facilities to support this approach. For example, functionality can be structured by developing a class hierarchy and by implementing procedures to decompose complex functionality.

Hierarchical CPNs, as presented in this section, have been formalized in [18,19] and implemented in CPN Tools. CPN Tools supports top-down and bottom-up design. Another approach for defining modules is to specify the module interface as a set of *transitions*. Flattening a hierarchical Petri net then corresponds to fusing equally labeled interface transitions, an established concept in the Petri

net literature. Whereas place fusion models asynchronous communication—that is, sending a token by one module is not synchronized by receiving this token by another module—transition fusion models synchronous communication. Synchronous communication is similar to invoking methods in object-oriented programming and is the dominant composition approach in process algebraic approaches [8,11]. Transition fusion is not supported by CPN Tools.

A different approach to introduce hierarchy in Petri nets is the *nets-in-nets paradigm* proposed by Valk [30]. Whereas traditionally tokens are passive, tokens in this paradigm can be active. One can think about such tokens as agents rather than data containers. As an agent has behavior, the token modeling this agent can represent a Petri net again. That way, the nets-in-nets paradigm supports the modeling of hierarchy. The Renew tool [21] supports the modeling, execution, and analysis of a particular instance of the nets-in-nets paradigm.

4 Simple CPN Patterns

In this section, we introduce some simple modeling *patterns* that can be used frequently. The patterns make it possible to model constructs not natively possible using the CPN formalism. As indicated in the introduction, the idea to provide patterns for modeling in terms of CPNs was first proposed in [26] where 34 patterns were identified. The patterns are briefly described in the Appendix of the paper. Subsets of these patterns can also be found on the CPN Tools Web page[2] and in books such as [6,15,18,19]. However, these publications do not explicitly identify and name these patterns.[3]

In this section, we look at patterns for bounding the number of tokens that can be in a place at once, for imposing an order of how tokens are added to and removed from places, for checking the number of tokens in a place, and for folding equal or similar net structures into one generic copy. In the next section, we present more advanced patterns to model more complex constructs that occur less often.

Unlike in [25,26], we will not be using a strict format for describing the patterns. Instead, we use examples based on the hierarchical gas station example from Fig. 6 with the environment module from Fig. 7(left) and the gas station module from Fig. 8. In our examples, we replace only the gas station module.

4.1 Bounded Places Using Complement Places

Figure 8 models the serving of a car as a single transition, Serve, indicating that this action is instantaneous. As it actually takes some time to serve a customer, this is an oversimplification. For this reason, we split this action into two actions: start serving and end serving. We then obtain the situation in Fig. 9(left).

[2] See `http://cpntools.org`

[3] Note that patterns refer to frequently recurring modeling problems and their solutions. Therefore, patterns are always based on earlier work and not intended to be original.

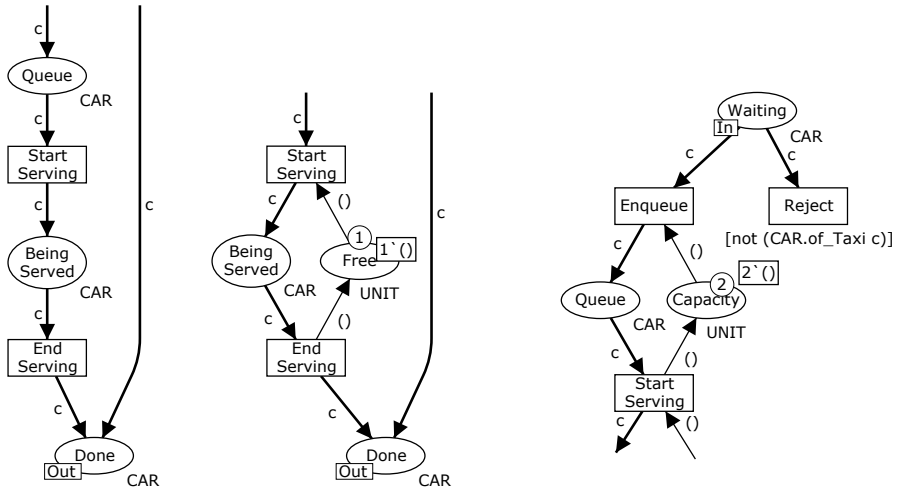


Fig. 9. CPN models of the gas station module with non-instantaneous serving of customers (left), with only one pump (middle), and with bounded queue size (right)

Here, we have a transition Start Serving, moving a car from place Queue to Being Served, and End Serving, moving a car from place Being Served to Done. The model now reflects that serving a customer is not an instantaneous action, but it introduces a new problem. Consider what happens if we have three cars in Queue. Now, we can Start Serving all three of them. This does not really make sense if the gas station has only one pump; rather, the cars should remain in the queue until the pump is free, at which point we start serving the next one in line. We thus need to bound the number of cars (tokens) that can be in place Being Served at any point in time. Therefore, we look at modeling patterns to limit the number of tokens in place Being Served.

The simplest way to bound the number of tokens in a place is to introduce a *complement place* (or anti place). The idea is to ensure an invariant, namely that the total number of tokens in this place and its complement place together is a constant. A place, which can at most contain a predetermined number of tokens, is *bounded*. A bounded place has a certain *capacity*. Bounded places are related to the concept of *safe places* in low-level Petri nets.

In Fig. 9(middle), we have added a new place Free with type UNIT. The type UNIT is declared to be:

```
colset UNIT = unit;
```

The unit type is a type which contains only one element, (), and simulates the behavior of (black) tokens of classical Petri nets: tokens of this type are indistinguishable, so we only care about the number of tokens. Whenever we produce a token in Being Served, we remove a token from Free, and vice versa. This ensures that the number of tokens in Being Served and Free remains unchanged and hence that Being Served contains at most one token. We can think of this

as consuming a right to produce a token in Being Served when we need it and returning the right when we no longer need it (when we consume a token).

In general, we model a complement place for a place by mirroring all arcs connected to the original place. That is, whenever there is an arc from a transition to the original place, we add an arc from the complement place to the transition; and whenever there is an arc from the original place to a transition, we add an arc from the transition to the complement place. The type of the complement place can be UNIT, as in the example, or it can be another type if we want to model a complement place for bounding multiple places or for bounding tokens of a particular value. For example, assume that we need to adapt in Fig. 9(middle) to model the situation that there are five pumps: two diesel pumps, two petrol pumps, and one pump for electric cars. In this case, the complement place initially has five tokens making sure that each pump can only be used for one car at a time. If the pump for electric cars is busy while the other four pumps are free, then it is impossible to serve another electric car. This can be ensured by using a different type for place Free, for example,

```
colset Fuel = with diesel | petrol | electric;
```

Moreover, type CAR needs to be extended to indicate the type of fuel a car requires and the arc inscriptions need to be adapted accordingly. However, the principle is the same: We consume a token from the complement place whenever we produce one token in the original place. Likewise, we produce a token in the complement place whenever we consume one token from the original place. In Fig. 9(middle), the transitions unconditionally produce/consume tokens in/from the original place, so we do the same for the complement place. In more advanced examples, we may produce/consume a varying number of tokens in/from the original place depending on the binding of the transition; that is, we need to make the expressions on arcs from/to the complement place reflect this.

The pattern to add a complement place to bound the number of tokens in another place is used frequently. Consider for example, locking in databases, kanbans in production systems, and message buffers in middleware.

4.2 Inhibitor Arcs Using Counter Places

A gas station needs to reject customers if it does not have the capacity to serve them; otherwise, they are added to the queue. Our model does not reflect that, as customers are rejected nondeterministically. Transition Reject should be enabled only if the Queue is full. As a first attempt, we add a complement place, Capacity, for place Queue and obtain the situation in Fig. 9(right). Now, the queue size is limited to 2. At this point, we start unconditionally rejecting customers. The model is still not correct, though, as we may also reject customers before the limit is reached. We thus want to disable Reject if Capacity contains more than zero tokens (i.e., if the queue is not full).

We can model such a condition in various ways. An arc that inhibits enabling of a transition if a place contains more than a specified number of tokens is an *inhibitor arc*. An inhibitor arc can in particular be used to test whether a

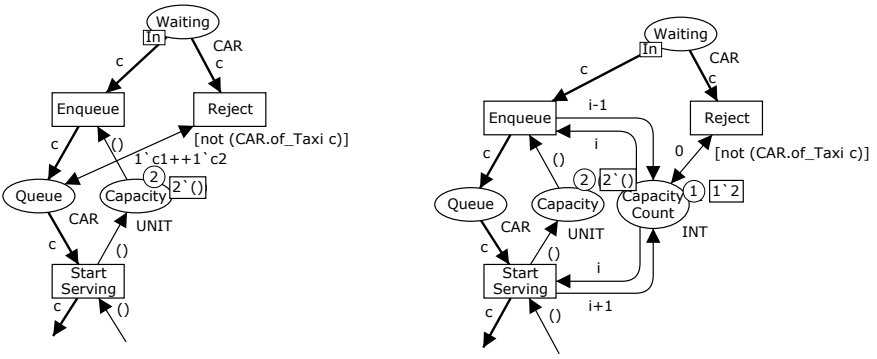


Fig. 10. Two gas station modules which reject customers only when there is no capacity for them

place contains zero tokens, which is also known as *zero-testing*. Some Petri net formalisms contain inhibitor arcs natively, but for CPNs, this is not necessary, as we can easily model them. The simplest way to model an inhibitor arc in our example is to add an arc between Queue and Reject checking that Queue contains two tokens, like in Fig. 10(left). This arc contains an arrowhead in both directions. This is a shorthand for an arc in both directions with the same expression. In CPN terminology, such an arc is called a *double arc* and in the literature, it is also known as a *test arc* or *read arc*.[4] The double arc between Queue and Reject allows us to test that two tokens are present in Queue without modifying them.

This solution works only for bounded places and is difficult to scale with the bound of the place (as we need a variable for each token). Instead, we introduce a general way of modeling inhibitor arcs here and another solution in Sect. 4.3. The basic idea of the general solution is to introduce a *counter place*. This place counts the number of tokens in a place. It is similar to a complement place, but we store the number of tokens as an integer rather than indistinguishable tokens.

In Fig. 10(right), we have added a counter place Capacity Count counting the number of tokens in Capacity. The type of Capacity Count is INT. This place is similar to place Next Car ID used in Figs. 1 and 7(left). However, now we increase the value whenever we add a token to Capacity (Enqueue) and decrease it when we remove one (Start Serving). The double arc between Reject and Capacity Count tests that the remaining capacity is 0; that is, cars are only rejected if no capacity is left.

Place Capacity is redundant after adding Capacity Count. Recall that Capacity was introduced as a complement place to bound the number of tokens in place Queue. We can also bound the number of tokens in place Queue by inhibiting the enabling of Enqueue when it contains more than one token—that is, by using a guard.

[4] There are subtle differences between test and read arcs depending on the exact transition execution semantics; we will not elaborate on this in this paper.

Fig. 11(left) shows the situation where Capacity is a counter place for place Queue; that is, the value of the token in Capacity corresponds to the number of tokens in Queue. Both Enqueue and Reject use this information to block if needed; that is, the arcs between Capacity and Enqueue update Capacity and serve as inhibitor arc. The guard added to Enqueue ensures that Enqueue is only enabled if the number of tokens in Queue is less than MAX_CAPACITY. MAX_CAPACITY is a constant defined in the declarations of the model as:

```
val MAX_CAPACITY = 2;
```

This declaration allows us to use a symbolic constant instead of writing the same value in multiple places, which improves the readability of the model and makes it easier to change the value of the constant if necessary. We have also added a double arc between Capacity and Reject and an extra clause to the guard of Reject checking that the value of the token is greater than or equal to MAX_CAPACITY. We use a comma (,) to separate clauses in the guard. This acts as a shorthand for logical and (which in CPN-ML is written as andalso).

4.3 FIFO-Places Using Complement Places or Lists

Any variant of the gas station we have looked at until now (see Figs. 7–11) serves cars in a random order. For most gas stations, this does not reflect reality; rather,

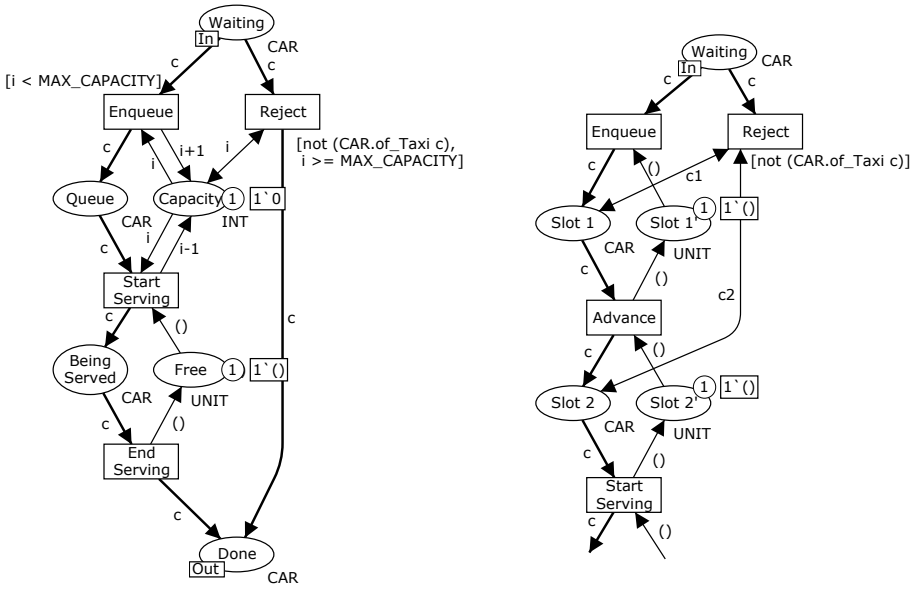


Fig. 11. Improved gas station module which only rejects customers when there is no capacity for them (left), and gas station module which serves customers in the order they entered the queue (right)

most gas stations serve customers using a first-come, first-served policy, and we want to make our model reflect this. A place from which tokens are removed in the same order as they are added is a *FIFO-place* (first-in, first-out place).

Our first idea is to model each location of the queue explicitly using a complement place to ensure that each location has at most one car. This is shown in Fig. 11(right). The only new thing is that we have a double arc between each location of the queue and transition Reject. These arcs test that every location in the queue is filled before rejecting customers. We have used separate variables for each double arc, as we are not requiring that each location contains the same car (which in this model is impossible). Each car arriving at Waiting can only enter the queue if the first spot is vacant. Likewise, a car in the first spot can only progress if the second spot is vacant. So as soon as a car has entered the queue, it is guaranteed to be served in the order it appeared. This is in contrast to place Waiting where cars are not ordered.

The queue in Fig. 11(right) serves its purpose but has some problems. First, the construction is not scalable. Compared to Fig. 11(left), where we could change the capacity of the queue by just changing the value of a constant, we have to add two places, a transition, and four arcs for each slot we want to expand the queue with. Second, it seems a bit excessive that a car has to drive through each spot in the queue explicitly even if it is the only car. Instead of using the net structure to express what is essentially a data-structure, we—for the first time in this paper—use that CPNs support *list* types. The idea is to represent the cars in Queue as a queue rather than a multiset. In CPN-ML, a queue is easiest modeled using a list.

Figure 12(left) shows an implementation of a queue using lists. It is easier to compare this module with the one in Fig. 11(left) rather than the one in Fig. 11(right). We have changed the type of Queue from CAR to CARS and use some more elaborate expressions on the arcs around the place. The type of Queue is a list of cars declared as:

```
colset CARS = list CAR;
```

Place Queue has an initial marking, 1‘[], indicating that the place contains a single token, an empty list (which is written as [] in CPN-ML). Whenever we produce a token in Queue in Fig. 11(left), we now, in Fig. 12(left), replace the token in Queue with a new list consisting of the previous list with the new element appended at the end. This is written as cs ^^ [c] in CPN-ML (i.e., append the singleton list [c] to the end of list cs). We have also added an arc opposite the original arc to get access to the previous value (cs). We need to introduce a variable cs that is declared as:

```
var cs: CARS;
```

Where we previously removed an arbitrary token from Queue, we now take the first element of the list and return the tail of the list to Queue. We do this by using pattern matching, which is a powerful mechanism CPN-ML [19] inherits from Standard ML [24]. The expression c::cs assigns the head of the list to c

and the tail to cs. A transition using such an expression on an input arc is only enabled when the list on the corresponding place contains at least one element.

The list structure also enables an alternative realization of the inhibitor arc pattern. As all tokens are inside a single data-structure, we can test the number of tokens without maintaining a separate counter. We have eliminated Capacity and changed the guards of Enqueue and Reject to refer to length cs, which returns the number of elements in list cs. Thus, if we have imposed an ordering of elements on a place, we can count the elements directly.

Another advantage of using the pattern in Fig. 12(left) is that we can use any data structure to impose any ordering of elements. For instance, we can change the inscription on the arc from Enqueue to Queue to c::cs to add the new car to the head of list cs, thereby implementing a *stack place* from which tokens are removed in last-in, first-out order. We can also implement *priority queue place*s by sorting the list according to a priority upon insertion. See [26] for concrete examples.

Figure 12(left) (but also the other CPN models modeling a queue for the cars waiting to be served) has the problem that tokens may be queueing in place Waiting. When the queue has reached its maximal capacity and a taxi arrives via port Waiting, then Enqueue and Reject are unable to handle the taxi. One can handle this in different ways. However, in case of a stochastic arrival process, it is impossible to ensure that there is a free position in the queue for taxis.

4.4 Folding Identical Net Structures

Most of the gas stations considered in this section (Figs. 9–12) had only one pump. We can easily change that by increasing the number of tokens initially in place Free, but only if we do not care which pump a customer uses. Now, assume that we add an extra step after refueling for paying for the fuel, as seen in the fragment in Fig. 12(right). This model does not preserve the information for which pump the customer has to pay.

It is possible to retain the information about which pump was used by duplicating the structure representing the pump and the payment procedure; a model doing so is shown in Fig. 13(left). Now, depending on which pump we chose to use initially, we have to execute either transition Pay 1 or Pay 2, thereby ensuring that each car pays for the gas it actually refueled.

Naturally, duplicating net structure is rarely the best solution. If the focus of the model is the geographical distribution of cars during a day, it may be a good choice as we have a one-to-one correspondence between places of the model and physical locations. Here, we are not interested in that, so it may be better to *fold* the two paths in Fig. 13(left) into one, thereby avoiding copying and making it easier to subsequently add more pumps or to change the behavior of all pumps (e.g., we may want to keep the reservation of a pump until the customer has paid, to make it even easier to match the pump and customer to the amount of gas purchased). A folded version of the model in Fig. 13(left) is shown in Fig. 13(right). In the remainder of this section, we explain folding using this example.

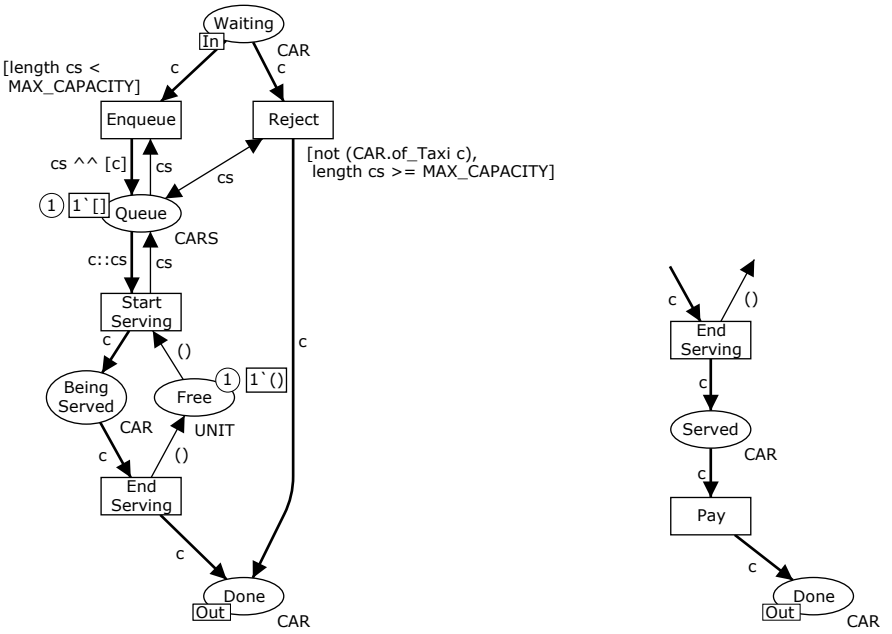


Fig. 12. Improved gas station module which serves customers in the order they entered the queue by using lists instead of the net structure (left) and gas station module which requires users to pay (right).

Folding means that we add to all places and expressions another component representing the identifier of the folded value. In our example, we define a new type, PUMP, and a variable of that type as:

```
colset PUMP = index pump with 1..2;
var p: PUMP;
```

An index type consists of a name (here pump) index by a set of integers (here 1..2); that is, allowed values of tokens in PUMP are {pump(1),pump(2)}. Consider using index types when you mathematically would have used index values like $pump_1, pump_2$. We then have to define types for all places we wish to fold (here Being Served, Free, and Served). We define these types as Cartesian products of the identifier used to fold and the original types. If the original type was UNIT, there is no need to keep it in the Cartesian product and we can just use the identifier type. We have replaced CAR on Being Served and Served with CARxPUMP and UNIT on Free with PUMP. Type CARxPUMP is declared as:

```
colset CxP = product CAR * PUMP;
```

This is the syntax for declaring a Cartesian product of preexisting types. We can also declare Cartesian products of more than two types by adding them at the end. We use the naming convention of using the original types (or the first letters) separated by a lowercase letter x, but sometimes it may be more useful

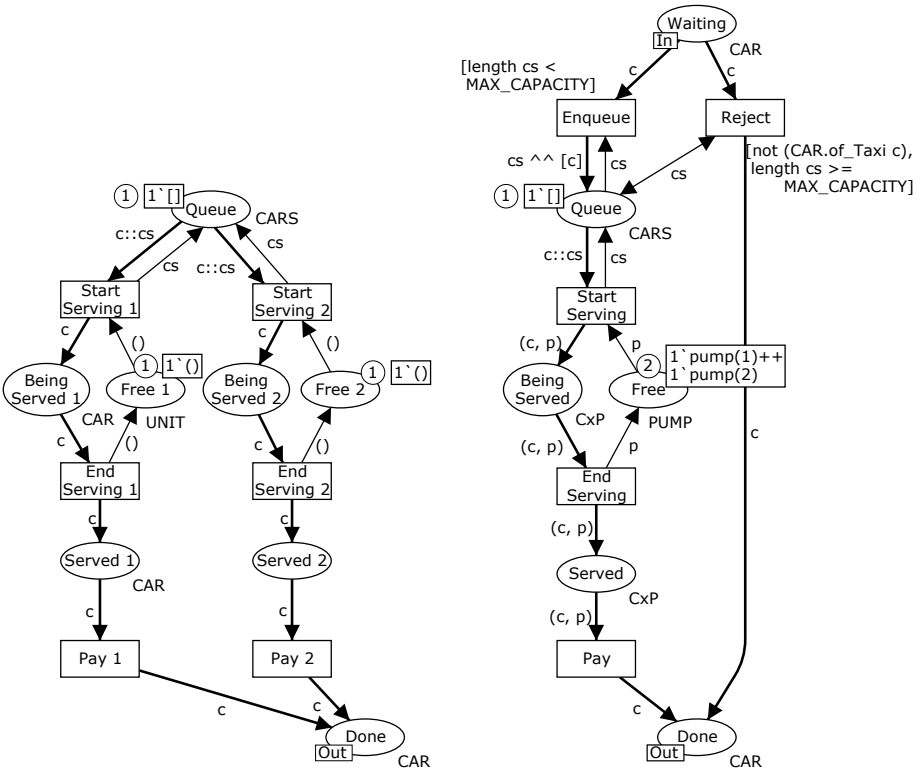


Fig. 13. Gas station modules which model the situation where payments are linked to the pump used. Two alternative modules are shown: with (left) and without (right) replicated net structure.

to give it a more descriptive name if the product has a meaning in the domain of the model. Changing the types of places also requires us to change the initial marking; in our example, this is simple, as only Free has a nonempty initial marking. We add an initial marking of 1‘pump(1)++1‘pump(2), specifying that initially both pump(1) and pump(2) are free for use. We also have to update all arc expressions. In Fig. 13(right), we just carry around the pump id and the car id, changing all inscriptions consisting of c to (c, p)—a pair of a car and a pump—and all inscriptions consisting of () to p, the id of the pump.

Figure 14 shows a fragment of a slightly modified version of the gas station, illustrating that we now only have to change the behavior once to change it for all pumps. In this version, a pump is occupied until a customer has paid. We have executed some steps of the model; for example, car Regular(7) is currently being served at pump(1), whereas car Regular(3) is done and is about to pay for the fuel taken from pump(2). There is no available pump at this time (i.e., Free contains no tokens), and car Regular(5) has been served and paid (or refused service) and is now done.

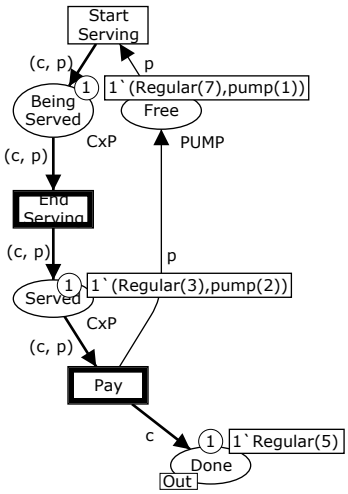


Fig. 14. Fragment of modified gas station module which requires users to pay for their own usage

In the examples we have seen here, we have treated each customer equally regardless of the pump they use, but we could also discriminate depending on the pump they use—for example, by inspecting the pump in the guard—introducing different paths depending on which pump a car uses. In fact, the entire model can be seen as a folded model in which the same procedure is shared for all cars. All cars are treated almost the same, except that taxis are never rejected, as seen by the guard of transition Reject. When we have a finite number of objects having the same behavior, such as the two pumps, it is convenient to share the net structure among them to avoid net replication. When we have an unbounded number of objects, such as all cars, it is not only convenient, but even necessary, as we would (theoretically) have to replicate the net structure an infinite number of times to be able to distinguish them.

5 Advanced Modeling Concepts

The patterns, we presented in the previous section, cover constructs that frequently occur in models of systems and processes. In this section, we present two additional patterns: message broadcast and region flush [25,26]. These patterns are more advanced than the previously presented patterns and cover constructs that occur less frequently in models. Before presenting these patterns, we introduce the concept of prioritized transitions, as supported by CPN Tools version 3.0, and illustrate how this concept can simplify the modeling of systems.

5.1 Extending Transitions with Priorities

A CPN may have a reachable marking in which several conflicting transitions are enabled. We refer to this situation as a *nondeterministic choice*. The marking in the gas station module shown in Fig. 15(left) is an example of a nondeterministic choice. The token in place Waiting models a waiting, regular car. This marking constitutes two enabled bindings, one enabling transition Enqueue and the second enabling transition Reject. Which of these two transitions fires is not predetermined. In reality, this would mean that an arriving car may be rejected even though the queue is not full yet. Because this situation is not desirable, we need to adjust the model such that in the situation shown in Fig. 15(left) always transition Enqueue fires. Earlier, in Sect. 4.3, we resolved this problem by adding an inhibitor arc between Queue and Reject (see Fig. 12(left)). In the following, we present another solution by prioritizing the firing of transition Enqueue over the firing of transition Reject.

Priority or *prioritized transitions* is supported in CPN Tools from version 3.0 and onwards. The modeler can assign a priority to each transition: *P_HIGH*, *P_NORMAL*, and *P_LOW*. The default value is *P_NORMAL*. Alternatively, it is also possible to specify the priority of a transition as an integer, where 0 represents the highest priority and larger numbers lower priorities. The built-in priorities, P_HIGH, P_NORMAL, and P_LOW correspond to the integer values 100, 1,000, and 10,000, respectively. The semantics of the model are defined by calculating first all enabled transitions ignoring priority and then considering only transitions with the highest priority. If there are multiple enabled binding elements having the highest priority, then one of them is nondeterministically chosen.

Figure 15(right) results from Fig. 15(left) by assigning value P_HIGH to transition Enqueue—all other transitions have the default value P_NORMAL, which is not explicitly shown in Fig. 15(right). As a result, only transition Enqueue is enabled in the marking shown in Fig. 15(right), because transition Reject has a lower priority than Enqueue.

The example illustrates the advantage of extending CPNs with priorities, namely simplicity of the model. The interplay of priorities is a *global* property

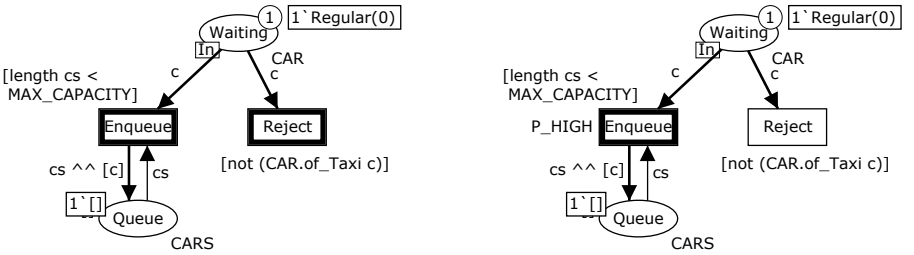


Fig. 15. CPN model without priority (left) and where Enqueue has a higher priority than Reject (right)

of a CPN; that is, assigning a priority to a single transitions may affect the enabling of all other transitions. Therefore, it is possible that the resulting CPN allows for undesired behavior. As an illustration, suppose we assign *P_LOW* to Reject and *P_NORMAL* to all other transitions. Having the lowest priority of all transitions, Reject can only fire if it is enabled (in the setting without priorities) and no other transition is enabled. However, this is never the case, because the environment module can continuously produce tokens in Waiting. As a result, Reject is dead.

Priorities simplify the modeling, but they do not increase the expressiveness of CPNs. Every CPN with priorities can also be modeled as a CPN without priorities. In Fig. 15(right), we can remove the priority from transition Enqueue if we use an inhibitor arc and extend the transition guard of Reject, as shown in Fig. 12(left). In general, expressing priorities between transitions without using the concept of transition priorities is nontrivial, in particular, if the transitions have disjoint presets.

5.2 Message Broadcast

Sometimes we need to model sending of a message to an unknown number of objects. Such a task is referred to as a *message broadcast*. It is particularly useful for modeling systems where many participants interact with each other (e.g., interorganizational business processes) and for message protocols. We illustrate this pattern with the following modification of the gas station. Suppose that there is a promotion at the gas station and every car driver who has refueled her car gets a voucher before paying for the gas. The difficulty is that we do not know the number of car drivers. The promotor would go to each car driver and hand over the voucher. Figure 16, which is an extension of Fig. 13(right), shows how we can model such a broadcast as a CPN.

Firing transition Start starts the procedure. First, we identify all car drivers who should receive a voucher. To this end, transition Read stores a copy of all tokens of place Served in a list in place Receivers. Each list entry refers to one car driver who will receive a voucher. The guard of transition Read ensures that the same token cannot be added twice to the list. Observe that this construction is independent from the number of tokens in place Served. By assigning a high priority to transition Read, we can make sure that Begin Broadcast can only fire if there are still tokens in place Served that have not been added to the list in place Receivers. When the list is complete, Begin Broadcast fires and the broadcast starts. Note that transition Pay cannot fire, because Pending is unmarked. Firing Begin Broadcast produces the length of the car list cs in Pending. Furthermore, as variable cs is a list of cars and Vouchers is of type CAR, cs is unfolded and for each list entry a single token is produced in Vouchers; that is, CPN Tools automatically converts the list cs into a multiset of tokens for place Vouchers. Transition End Broadcast models the distribution of the vouchers to each car driver. After all car drivers have received a voucher, there is a token with value 0 in place Pending, thus enabling Pay and Start. If Start fires again, then every car driver receives a second voucher.

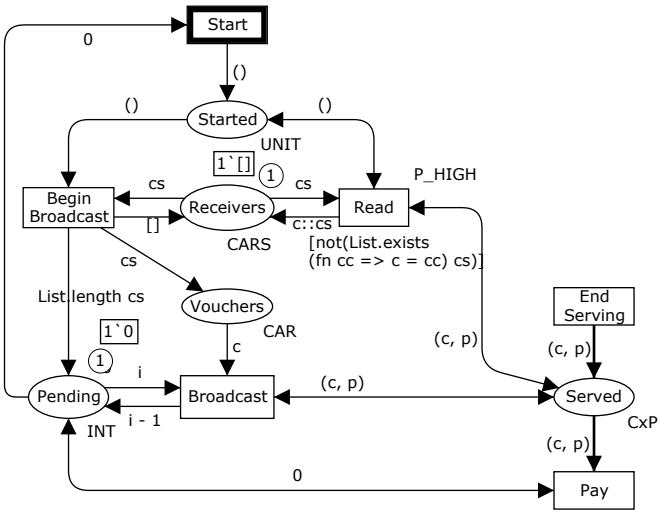


Fig. 16. Broadcasting vouchers to all car drivers who refueled their cars

5.3 Region Flush

Suppose that the gas station attendant is eager to stop working in time. Every day at 6 p.m. he serves only the cars that are being refueled; all cars in the queue will not be served anymore and have to drive on. To model this as a CPN, we must move all tokens from place Queue to place Done and also reset the value of place Capacity to 0, thereby making sure that transition Start Serving does not fire while tokens from Queue are moved to Done. This is trivial if we have modeled the queue using lists as in Fig. 12(left). However, when modeling the waiting cars in the queue as individual tokens, things become more involved. Figure 17(top) extends the model in Fig. 11(left) with this functionality. Transition Close models the closing of the gas station. It removes the token from place Capacity. This token is needed to learn the number of cars in the queue and to prevent transitions Enqueue and Start Serving from firing. After the queue has been flushed—that is, all cars have been moved to Done—transition Open models the opening of the gas station by initializing place Capacity again.[5] Only then transitions Reject and Start Serving can become enabled. In case we need to flush more than one place, we must copy the pattern accordingly.

Removing tokens from a part of a CPN is referred to as *region flush*. The idea is to disable the transitions in the respective part (i.e., the region) of the CPN while removing all tokens from the places in the region. Afterward, the region is optionally reset.

[5] When adding explicit time, we can model that the gas station opens again at a particular time (e.g., 9 a.m. the next day). Transition Open is not supposed to fire immediately after closing the gas station; this needs to be governed by the time concept explained in the next section.

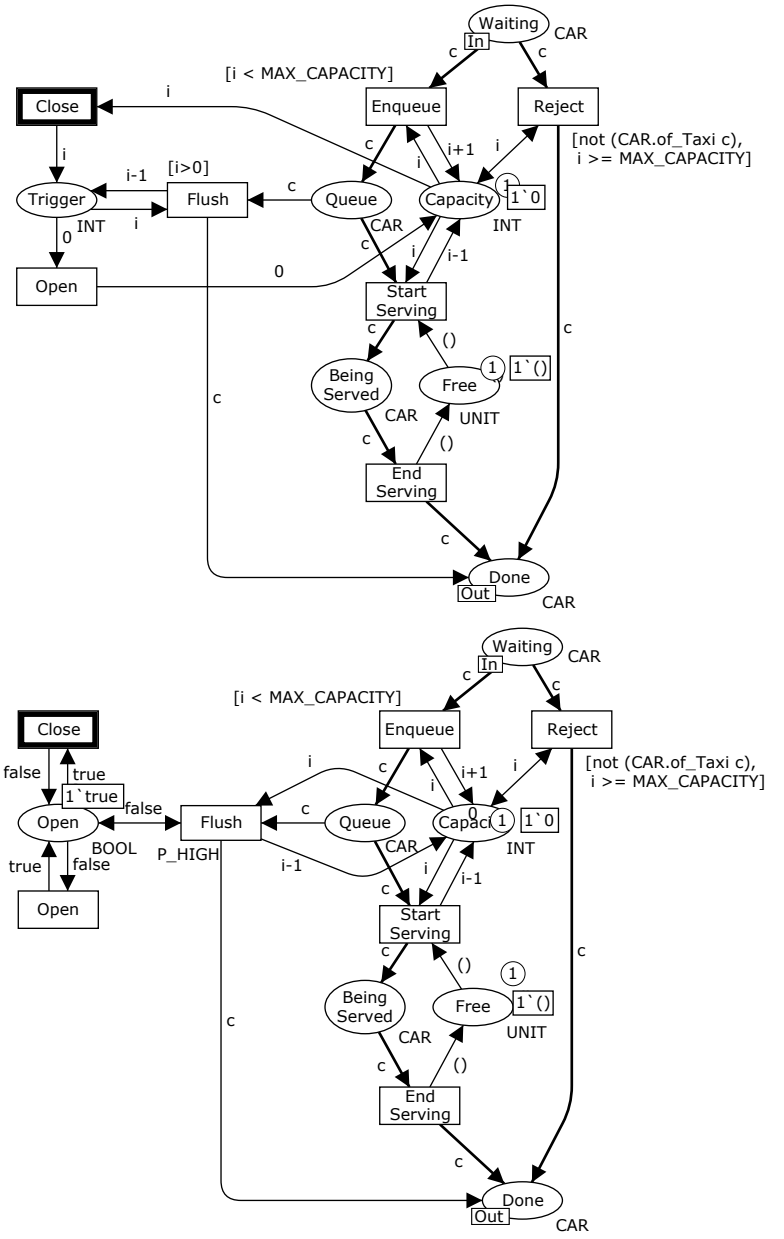


Fig. 17. Removing all cars from queue without (top) and with (bottom) priority

Using the concept of prioritized transitions, as introduced in this section, we can generalize the model of a region flush. The respective CPN model is shown in Fig. 17(bottom). Transition Flush reads a Boolean false from place Open, moves a car token from place Queue to place Done, and decrements the value of Capacity. As only Flush has a high priority, it can fire until the queue is empty without being in conflict with any other transition. Transitions Close and Open model the closing and opening of the gas station. Close produces a Boolean false in place Open. If there is at least one car in the queue, then Flush is enabled (because of its priority). Only after the queue has been flushed, transition Open can change the Boolean in place Open to true, modeling that the gas station opens again.

The advantage of using the pattern with priority in Fig. 17(bottom) is that it is independent of the presence of complement place Capacity. In contrast, Fig. 17(top) relies on place Capacity as it requires knowledge about the number of cars to be removed; otherwise, we would not know when we have moved all waiting cars to place Done and, hence, when we can open the gas station. Place Flush is connected to place Capacity to update this complement place; it is not used to see how many cars need to be flushed.

In Fig. 17, only one place is flushed (place Queue). The construct needs to be copied per place. This may become quite involved, therefore, workflow languages such as YAWL support this natively (see the cancellation pattern in [4,32]).

6 Modeling Time

Time is an important aspect in many systems. Let us, for example, consider the gas station obtained by combining the modules in Figs. 6, 7(left), and 13(right) (shown together as Fig. 18). In this example, we have modeled the serving of customers as two transitions, Start Serving and End Serving, indicating that the action of serving is not instantaneous. However, serving customers is an atomic action in the sense that neither the car nor the pump can be used for anything else during the feat. It would be more elegant to model serving as an atomic action which takes time. The action of paying for the gas should also take time. In this section, we look at how CPNs enable us to model timed aspects of systems and use this to extract information about the performance of the system. As shown in the next section, one can do experiments with a timed system using the model, thereby extracting performance data that can be used as input for decisions regarding the modeled system. For example, simulation can be used to find out whether it is better for a gas station to acquire extra pumps or to reserve more capacity for the queue if the number of customers doubles. We also look at the difference between real and integer time stamps and the interaction between timed models and prioritized transitions.

6.1 Time Basics

In CPNs, time is introduced by assuming a *global clock* representing the current *model time*. We assign a time stamp to each token. A time stamp on a token

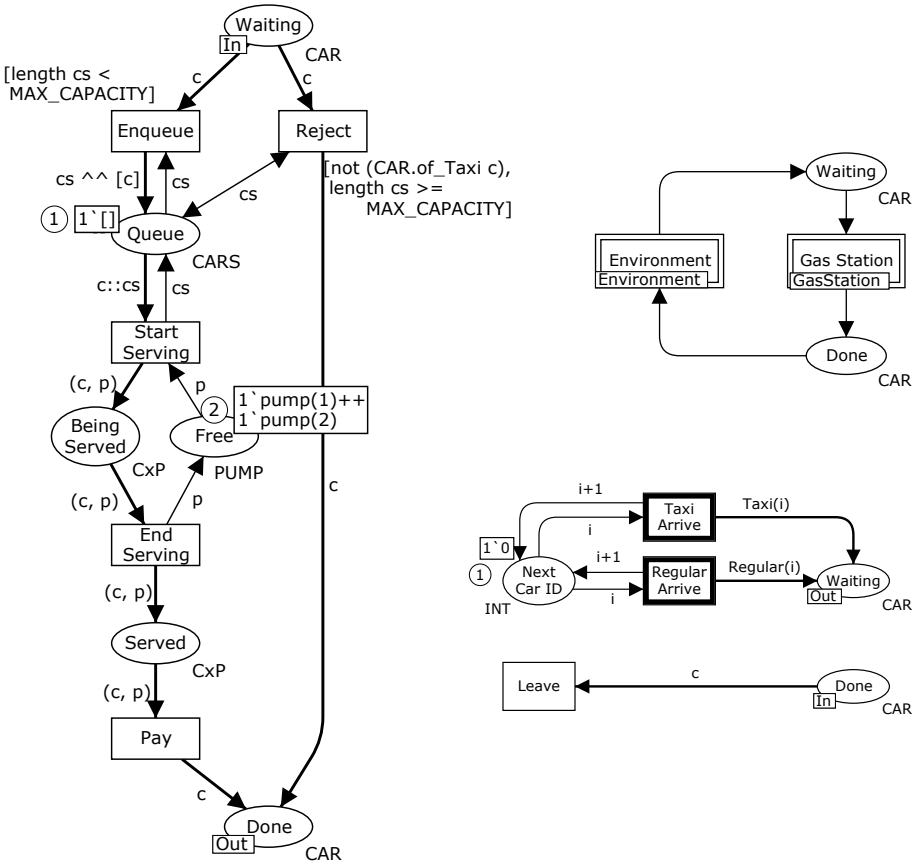


Fig. 18. Untimed gas station

indicates when the token can be consumed. A token can only be consumed if its time stamp is less than or equal to the current model time. In a sense, a token with a time stamp in the future (compared to the current model time) can be regarded as a promise that, at some point in the future, a token will be produced. Hence, one can think of time stamps as reversed expiry dates: tokens with a time stamp x can be consumed at time x or later.

Figure 19(left) shows a timed version of the gas station from Fig. 18. We have merged the two Serve transitions into a single one and added an annotation @+5 to transition Serve. This annotation specifies that executing the transition takes 5 units of time (i.e., firing is atomic, however, the tokens are produced with delay of 5 time units). In the same way, we have added an annotation to Pay that states that paying takes 2 time units. We also see that the current marking of Waiting, Pumps, and Paying reflects time stamps, so that Regular(1) in Waiting is available at time 0 (due to @0 after the token value). Furthermore, we now join tokens with different values using +++ instead of ++. This is merely a technicality due to typing. We see that Taxi(0) has been served by pump(2) (it is on Served)

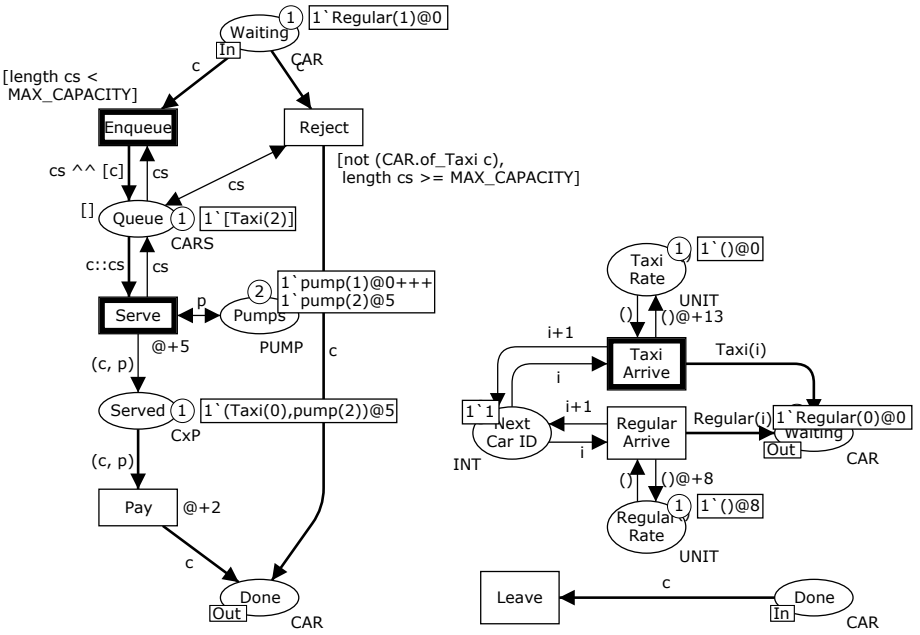


Fig. 19. Simple timed version of the gas station (left) and environment (right)

and that Taxi(2) is currently in line to be served.[6] As pump(0) is available at time 0, the current model time is 0, and Serve is enabled at this time. Pay is not enabled at time 0, however, as the token in Served is not available until time 5. In a sense, the tokens (i.e., (Taxi(0),pump(2)) in Served and pump(2) in Pumps) have not been produced yet; we have only a promise that in the future (in 5 time units) the tokens will be produced.

Each pump can serve at most one car every 5 time units; that is, the time-related annotations specify that serving a customer takes time and during that time the pump is not available.

We have to specify for each type whether it is timed or not (we allow tokens without a time stamp, which is a shorthand for a token that is always available). We specify that a type should include a time stamp by adding the keyword timed at the end of the declaration. In our example:

```
colset CAR = union Regular: INT + Taxi: INT timed;
colset PUMP = index pump with 1..2 timed;
colset CxP = product CAR * PUMP timed;
```

The model of the environment in Fig. 18 does not take time into account and, therefore, produces cars that all appear at time 0. This causes all cars to enqueue

[6] Although there is a token referring to Taxi(0) in place Served, the service has not been completed yet (in fact, it just started). This can be seen by comparing the time stamp of the token in place Served (@5) with the current model time (0).

at this point. Thus, time will never progress, as time only progresses when there are no more enabled transitions at a given time stamp. We therefore need to make the environment time-aware. We would like that cars arrive at a constant rate, but that the rate is different for taxis and regular cars. We could add separate counters for each of the transitions and use a construction similar to the one used for Serve and Pump, but instead we choose to add separate places with a single token limiting the rate of the transitions. We use a timed UNIT as type and place a single token in each place, obtaining the environment in Fig. 19(right). We have not added time annotations to the transitions, but rather to the output arcs. This enables us to produce tokens that are available at different times. In Fig. 19(right), we have just executed Regular Arrive at time 0, yielding a car Regular(0) on Waiting which is available at time 0, and a token () on Regular Rate which is available at time 8. In this way, we have modeled that arriving takes no time, but the transitions can only occur at a specified rate. The rate for regular cars is once every 8 time units, and the rate for taxis is once every 13 time units. The token in Next Car ID is untimed and does not influence the enabling of any transitions as it is always available.

6.2 Embedding Time Stamps in Tokens

Place Queue in Fig. 19(left) always contains one token (the queue) and this token is always available as the place is untimed. Hence, the @+5 annotation of transition Serve does not apply to this place. In this particular situation, this is just fine; time progresses only in-between subsequent arrivals and because it takes time to serve a car. Although the untimed queue works well in this situation, there are situations in which an untimed queue cannot express the desired behavior—for example, if the transition Enqueue takes time or to model more complex priority and resource allocation rules, where the next time stamp needs to be computed based on the entire queue. Therefore, we would like to make the tokens of the queue timed as well, but this is not possible, because whereas we think of the queue as a list of tokens, the list is a single token and can therefore have only one time stamp. To fix this, we need to embed the time stamps of the tokens in the queue. We do not have access to the time stamp of a token (it is either available or not), but we do have access to the current model time, and can use that to embed time stamps in the elements of the queue, obtaining the model in Fig. 20. The idea is to not just store cars in the list representing the queue, but a pair of the car and the model time the car arrived; that is, we define the types:

```
colset CxT = product CAR * STRING;
colset CARS = list CxT timed;
var t : STRING;
```

Now, CARS is timed. We want the time stamp of the list token modeling the queue to be the minimum of all time stamps *in* the queue. If the queue contains no cars, we assign the current time to the queue. We need to convert the model

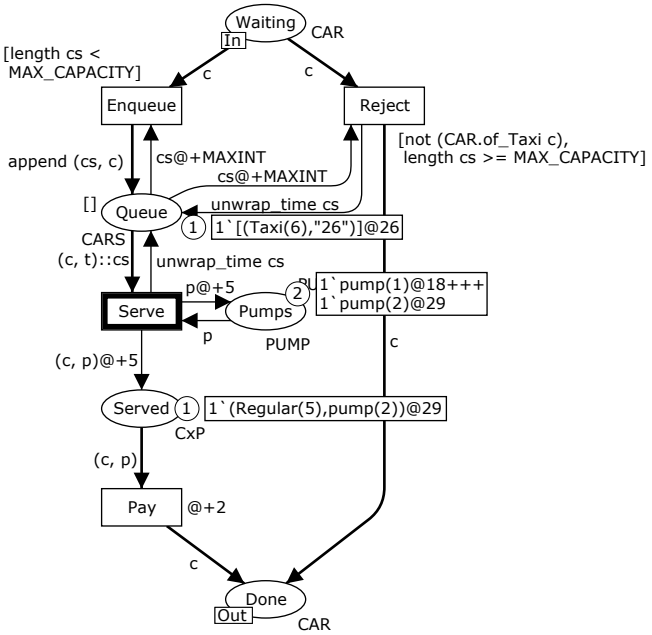


Fig. 20. Timed version which embeds time stamps in the list

time to and from strings, as there is no type in CPN Tools that directly corresponds to the type of the model time. For this purpose we define the following function:

```
fun modelTime() = ModelTime.toString(ModelTime.time())
```

Moreover, we need to compute the time stamp of the queue. Therefore, we define function unwrap_time, which takes as list of cars and returns the same list, but with a time stamp equal to the time stamp of the first car in the list.

```
fun unwrap_time([]) = [] @ (ModelTime.time())
  | unwrap_time((c, t)::rest) =
                ((c, t)::rest) @ (ModelTime.maketime t)
fun append(cs, c) = unwrap_time(cs ^^ [(c, modelTime())])
```

Function append adds a new car to the end of the list together with its arrival time converted to string format. Moreover, using unwrap_time the time stamp of the whole list is computed.

6.3 Ignoring Time Stamps of Tokens

After adding a time stamp to Queue, we need to make some minor changes to other transitions surrounding it so they are not wrongly delayed. First, we have changed Serve to no longer have a time inscription, but to instead only

delay the pump and car moved to Served, as unwrap_time takes care of delaying the queue the correct amount. If we kept the time inscription, the time stamp of the queue might also be delayed 5 time units, resulting in a car not being served immediately even though a pump is available. Second, we have added time inscriptions to the arcs from Queue to Enqueue and Reject. The inscription uses MAXINT, which is a constant, declared as:

```
val MAXINT = Option.valOf (Int.maxInt)
```

An inscription on an input arc means that the token can be consumed the indicated time units before it is really available, so consuming it MAXINT time units before basically says to ignore the time stamp. This allows us to reject or enqueue a car even though the queue is not currently available. Although transition Serve needs to wait until the first car is available, the queue can be updated at any time. In this situation, this is not required, but the example shows that we can have full control over time even when objects are put in a list.

6.4 Real Time and Random Distributions

Until now we have used constant delays and rates on all transitions. This is not realistic. Often we would rather know that something has an average delay or rate and perhaps even a guess (or assumption) about how the values are distributed. We thus wish to replace the delays and rates by values resulting from drawing a random number using a given distribution. However, most random distributions occurring in practice, especially randomly distributed times, are not integers. In previous versions of CPN Tools, we would need to scale the random values to a desired precision and convert the randomly drawn values to integers. But from version 3.0 and onwards, CPN Tools supports using real time stamps. Changing time stamps to be real values, we can create a new environment as in Fig. 21. We have changed only the inscriptions on arcs indicating the rates. Rather than using a constant value, we draw values from the exponential function, which randomly samples values from a negative exponential distribution, which is appropriate for modeling arrival rates. The exponential function is given a parameter λ, and it draws values with a mean value of λ^{-1}, which is why we write 1.0/13.0 rather than 13.0. We use inscription @++ rather than the inscription @+. This is needed when we want to make increments that are not integers. Figure 21 shows that the tokens now have time stamps that are not integers.

We would naturally like to make an implementation of the gas station also using randomly generated delays. Figure 22 shows an example. We have changed the time inscription of Pay to randomly draw a value from the normal distribution with a mean value of 2 and a variance of 1.[7] Transition Serve has become a bit more complex, as we want the car and pump to complete at the same time, so we cannot just add two calls to normal, as that would draw a random value for each. Instead, we have added a *code segment* to Serve, which outputs

[7] Note that the normal distribution is not a very suitable delay distribution as it may generate negative values that are effectively treated as zero's, thus shifting the mean.

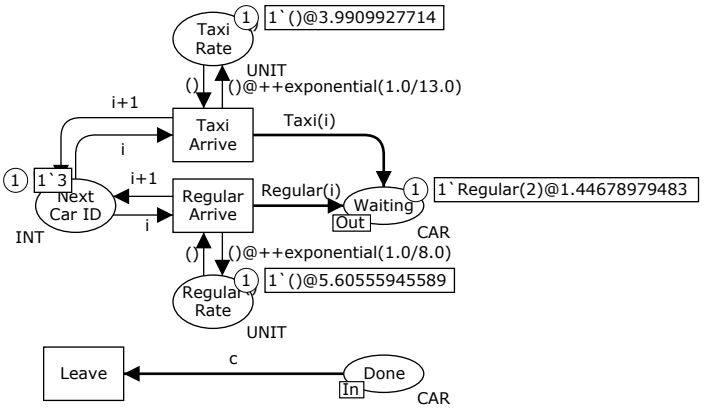


Fig. 21. Real timed environment drawing rates from the negative exponential distribution

a variable d and draws a value from a normal distribution with mean 5 and variance 2, converting it to a string (as variables cannot have type time). The time inscriptions use variable d and convert it into a time stamp.

6.5 Time and Priorities

The interplay of priorities on transitions as explained in Sect. 5.1 and time as explained earlier in this section is nontrivial, as they are both global properties. The basic rule is that *time takes precedence over priorities*, so if a low-priority transition is enabled at time 12, but a high-priority transition is enabled at time 15, the low-priority transition will be executed first. The intuition of the interplay of transition priorities and time is that simulation progresses along a time-line. At each point in time, we evaluate which transitions are enabled and execute the ones with the highest priority first. We then execute all transitions enabled at the current model time in a highest priority first order until no more transitions are enabled, at which point we increase the current model time and start anew.

In the gas station example, if we combine the model in Fig. 15 with the one in Fig. 19(left) (i.e., we add time stamps to the model in Fig. 15), we obtain the model in Fig. 23. Here, Served is enabled even though there are enough tokens for Enqueue to be enabled. The reason is that Regular(4) arrived at time 16, and the current model time is 16, at which point Serve is enabled (if tokens were enabled at the time, it would be enabled already at time 0 for pump(1) and at time 13 for pump(2)). Enqueue is not enabled until time 26, when Taxi(5) becomes available. So, even though Enqueue has high priority and all available tokens, it is preempted by Serve, which is enabled earlier.

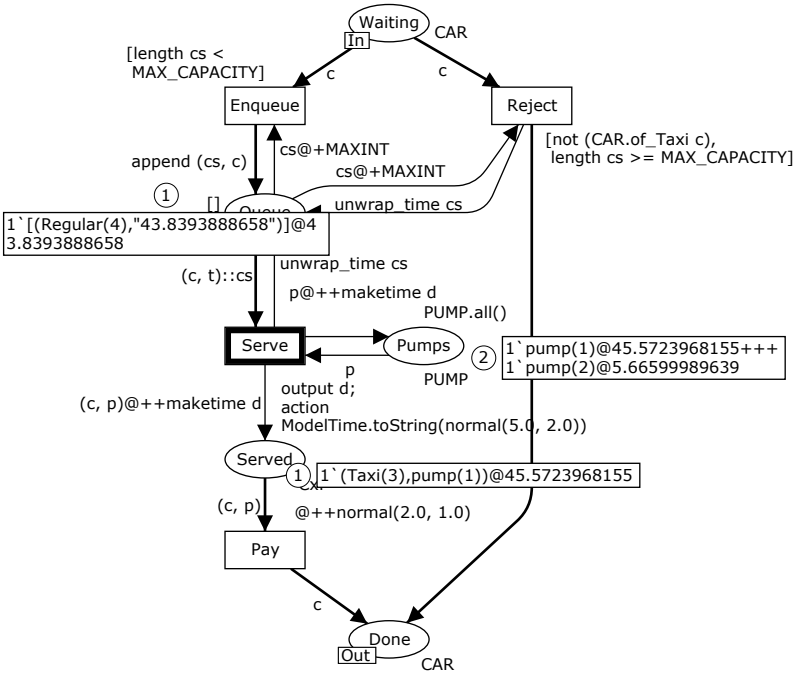


Fig. 22. Real timed gas station drawing average times from a normal distribution

6.6 Different Timing Concepts

Many different timing concepts have been introduced in the literature. Authors associate time to transitions, places, or arcs. In most timed Petri net models, transitions determine time delays. In only a few models, time delays are associated to places or arcs. Independent of the choice where to put the delay (i.e., transitions, places, or arcs), several types of delays can be distinguished, for example, deterministic (the delay is fixed), nondeterministic (the delay is a non-deterministic choice over an interval [1,23]), and stochastic (the delay is sampled from some probability distribution [22]). Even when the location of the delays and the type of delay are determined, there are still many possibilities. Adding time to Petri nets requires a redefinition of the enabling and firing rules. For example, when time is associated to transitions and delays are stochastic, one can use preselection semantics (i.e., first the transition to be fired is selected and only then the delay is determined) or race semantics (i.e., transitions are competing for tokens and the transition that finishes first takes the tokens). In the later case one needs to select a memory policy (age memory, enabling memory, or reset memory). This illustrates there are many ways to add time to Petri nets.

Generalized Stochastic Petri Nets (GSPNs) [9] are probably the most widely used Petri net model focusing on the time extension. This model allows for two types of transitions: transitions that do not take time (immediate transitions)

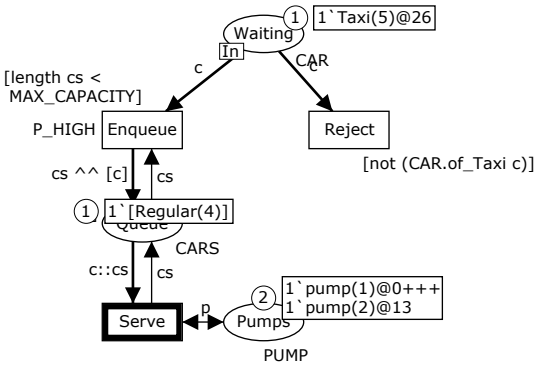


Fig. 23. Gas station with priorities and time stamps. The current time is 16

and transitions that have an enabling time that is sampled from a negative exponential distribution. Due to the memoryless property of the exponential distribution and the race semantics, it is possible to convert a GSPN into an embedded Markov chain, thus allowing for all kinds of analysis [22].

Most of the timed Petri net models described in literature have been tailored towards a particular analysis technique. Unfortunately, Petri net modeling languages using stochastic delays tend to impose restrictions to allow for Markovian analysis. Most Petri net modeling languages using fixed times or time bound by intervals also impose restrictions to allow for model checking.

CPNs aim at the modeling of large and complex processes. As shown in earlier sections, this requires tokens to be colored. Places may have types that cannot be enumerated (e.g., lists). In fact, Markovian analysis and model checking are unrealistic for applications that use the patterns described earlier; the state spaces of such models are simply too large to allow for exact analysis. Therefore, we need to resort to simulation. As a result, we do not have to put any restrictions on time and, therefore, we can select the most convenient and intuitive time extension.

Since tokens already have a value, it is most natural to also attach time stamps to tokens [1,15,18]. Since time inscriptions can be put on both input and output arcs and it is possible to inspect the current time (see Fig. 22), the designer has full control over the time in CPN Tools. Any of the timed Petri net models described in literature can be emulated. However, performance analysis is restricted to simulation.

7 Simulation

As indicated in the previous section, we need to resort to simulation to analyze the performance of complex processes. Therefore, we focus on this type of analysis. First, we position simulation in the broader spectrum of analysis techniques. Then, we show how to construct simulation models and how to monitor them.

Finally, we explain how to interpret the results and illustrate this by comparing different redesigns for our running example.

7.1 Overview of Analysis Techniques

The idea of simulation is to take an executable model (e.g., a CPN) and let it run several times. Each run can be seen as an experiment and corresponds to a "random walk" in the state space of the model. Even if it is impossible to construct the entire state space, it is possible to do such experiments; just "play the token game" repeatedly. A model may allow for infinitely many scenarios, that is, possible runs. Because only a finite number of scenarios can be executed, only a fraction of the entire state space can be explored. Consequently, simulation can, unlike verification, be applied to verify only the presence of errors and not their absence.

Verification of CPNs is challenging. Because of the introduction of data and time, the state space tends to be large, if not infinite. The only way to use model checking techniques effectively, is to limit the use of data and time. Typically, one needs to abstract from time and use types with a limited number of possible values. We refer to other papers in this ToPNoC volume that focus on this topic and that introduce techniques such as reachability graphs, coverability graphs, unfoldings, invariants, siphons, and traps.

Simulation is widely used for performance analysis. Performance analysis tries to make predictions about key performance indicators, such as response time and flow time, and to detect possible bottlenecks. The goal is to understand the *as-is* situation and to compare this with possible *to-be* situations.

Lion's share of Petri net research has focused on *model-based analysis*; that is, by analyzing a model, one hopes to be able to make meaningful statements about an as-is or to-be situation. However, such analysis only makes sense if the model reflects reality. Simulation results are irrelevant if the model has little in common with reality. Therefore, we advocate the use of *process mining* techniques in case event logs are available. Process mining [2] aims to discover process models from example behavior captured in different data sources (e.g., databases, transaction logs, and audit trails) and to relate existing models to such behavior. As shown in [27], it is possible to discover CPNs from events logs. Such CPNs model the control-flow of cases, resources, resource allocation, dataflow, routing probabilities, and routing conditions. In the context of workflow management systems, it is even possible to upload the current state of such a system into a CPN model and conduct *short-term simulation* [28]. Unlike classical simulation approaches that focus on steady-state behavior, the goal of short-term simulation is to make predictions about the near future to answers questions such as "How many orders will we have in the pipeline next week?", "What is the expected average response time tomorrow?", and "What will be the average flow time by the end of next week if I temporarily add two workers?". The focus of short-term simulation is on the transient behavior. This allows for a "fast forward button" into the future.

In the remainder of this section, we assume that we are able to create a CPN model that adequately reflects reality. Moreover, we focus on the steady-state behavior of processes. Nevertheless, we encourage the reader to consult [27,28] for more information about the alignment between reality and simulation models.

7.2 Adding Stochastic Behavior to a CPN

In Sect. 6 we showed how to add time to models. In our running example, cars are generated using a Poisson arrival process. This means that the time between two subsequent arrivals is sampled from a negative-exponential probability distribution. In any situation where there is a large population of potential entities that can generate requests (e.g., customers refueling their cars), a Poisson arrival process is most natural. One can show that if these entities are in steady state and independent, their behavior will always resemble a Poisson arrival process. To model the time it takes to refuel a car, we also need to use a probability distribution. Earlier, we used the normal distribution. CPN supports many probability distributions suitable for modeling time durations. Table 1 shows some examples.

Table 1. Random distribution functions

Function	Description
uniform(a:real,b:real):real	For $b > a$, uniform(a,b) samples a value from a uniform distribution with mean $(a + b)/2$.
exponential(r:real):real	For $r > 0$, exponential(r) samples a value from an exponential distribution with mean $1/r$.
erlang(n:int,r:real):real	For $n \geq 1$ and $r > 0$, erlang(n,r) samples a value from an Erlang distribution; that is, the sum of n independent exponentially distributed values with parameter r. The expected value is n/r.
normal(n:real,v:real):real	For $v \geq 0$, normal(n,v) samples a value from a normal distribution with mean n and variance v.

Let us now revisit our running example. Figure 24 shows another variant of the gas station. At the highest level, we now distinguish between cars that were served (place DoneS) and cars that were rejected (place DoneR). We use a Poisson arrival process to generate cars. The mean time between subsequent arrivals of taxis is 12 minutes. Regular cars arrive, on average, every 6 minutes. Hence, on average, $5 + 10 = 15$ cars arrive per hour. In our initial situation we have only one pump; that is, FreePumps contains only one token. We split the service transition into a StartServe and EndServe transition to explicitly show when a pump is busy or free. Transition Pay is still atomic and is an "infinite server"; that is, the transition takes time, but cars do not need to wait for one another.

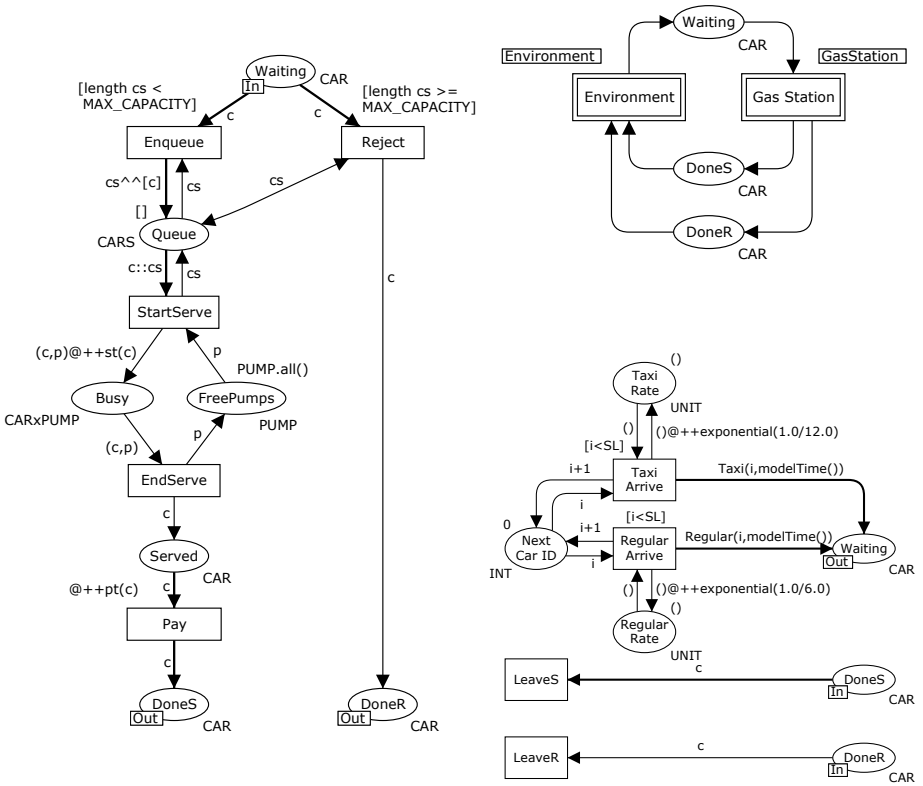


Fig. 24. The initial CPN used for simulation. Taxis and regular cars arrive using a Poisson process. The mean time in-between two taxis is 12 minutes. The mean time in-between two regular cars is 6 minutes. The service time and the time it takes to pay are sampled from a uniform distribution.

Figure 24 uses the following types:

```
colset STIME = string;
colset BCAR = product INT * STIME;
colset CAR = union Regular: BCAR + Taxi: BCAR timed;
colset CARS = list CAR;
colset PUMP = index pump with 1..1 timed;
colset CARxPUMP = product CAR * PUMP timed;
```

The types are self-explanatory, except for the addition of STIME. Every car has a unique ID of type INT and a creation time of type STIME. We use function modelTime defined earlier to inspect the global clock and convert the current type to a string such that it can be stored in a token. The creation time of a car has been added to measure flow times; that is, we measure the difference in time between the moment a car leaves module Environment and when it returns.

Additional declarations used in Fig. 24 are:

```
val MAX_CAPACITY = 3;
fun st(c) = uniform(2.0, 5.0);
fun pt(c) = uniform(1.0, 2.0);
val SL = 100000;
```

In our initial simulation model, we allow for 3 cars to queue. The time it takes to refuel a car is between 2 and 5 minutes (uniform distribution). The time it takes to pay is between 1 and 2 minutes (uniform distribution). For each simulation run, we create 100,000 cars. This is reflected by parameter SL used in module Environment.

7.3 Monitoring a CPN

Figure 24 only models the process we would like to analyze without modeling the measurements required for analysis. However, to extract simulation results, this is not sufficient; we need to indicate what kind of results should be collected (e.g., flow times, utilization, costs). In general, we would like to avoid "polluting" the model with extensions to collect statistics. This is why CPN Tools offers the possibility to add *monitors*. The idea of a monitor is to collect data from markings that are reached and bindings that are enabled during the simulation runs. For example, a *Marking size* monitor counts the average number of tokens in a place. In Fig. 24, we add two such monitors; one for place FreePumps and one for Busy. The average number of tokens in Busy divided by the number of pumps, indicates the average utilization of these pumps. To measure the average number of cars queueing, we add a *List length data collection* monitor for place Queue.

These monitors can be added without any effort. Just select the desired monitor and attach it to the corresponding place.

Measuring the flow time requires a bit more effort. We added the creation time of a car to the type CAR. In addition to an ID, a car also has a value of type STIME indicating the time it was created by the Poisson arrival process in module Environment. Transition LeaveS in module Environment consumes cars that have been served (see Fig. 24). This transition fires the moment the car has been refueled and payment has been completed. Hence, the difference between the time LeaveS fires and the time stored in STIME field of the car token is the flow time of a car. Obviously, we are interested in this duration. Later, we explore various alternatives and analyze their effect on the average flow time of cars that have been served. Using a *Data collection* monitor, we can measure the flow time. This monitor simply stores measurements. In this case, we need to specify that we are interested in the difference between the creation time and current time. Subsequently, these measurements are used by CPN Tools to calculate statistics such as average, variants, upper bounds, and lower bounds.

Cars that have not been served have a flow time of 0. Therefore, we do not need to measure the flow time for such cars. However, we want to measure the fraction of cars that is rejected. Again we use a *Data collection* monitor. However, now we do not measure the flow time, but measure whether a car was rejected

(record a value 1) or not (record a value 0). Hence for each car we record 0 or 1. By computing the average over these values, we know the fraction of cars that was rejected in the simulation run.

7.4 Interpreting the Results

After extending the simulation model shown in Fig. 24 with monitors, we can execute a simulation run in which 100,000 cars are generated (see parameter SL) and inspect the simulation results. For a particular simulation run, we find that:

- The average length of the list token in Queue is 0.915835;
- The mean number of tokens in Busy is 0.802938;
- The mean number of tokens in FreePumps is 0.197062;
- The average flow time of served cars is 9.001130 minutes; and
- The fraction of cars rejected is 0.089430.

Hence, the utilization is approximately 80% and on average about one car is waiting to be served. The average flow time is approximately 9 minutes and about 9% of the cars is rejected.

Based on one simulation run, we cannot make any conclusions. In another simulation run, the previous results could be different. Therefore, we need to compute *confidence intervals*. By repeating the simulation experiment several times, we can get an idea of the reliability of the results. Suppose that we repeat the experiment 10 times and measure the flow time for each simulation run. If the 10 values are close to one another, then we can be confident that the result is reliable. If the 10 values are far apart, then this is an indication that more or longer simulation runs are needed. Using standard statistical methods, one can compute confidence intervals based on subruns. For an introduction to subruns and confidence intervals in the context of CPNs, we refer to [6]. Here, we just show the results.

If we compute confidence intervals based on 10 simulation runs in which 100,000 cars are generated (i.e., the behavior of 1,000,000 cars is analyzed), then we get the following 90% confidence intervals:

- The average length of the list token in Queue is 0.900 ± 0.006; that is, with 90% confidence the average is between 0.894 and 0.906;
- The mean number of tokens in Busy is 0.798 ± 0.001; that is, with 90% confidence utilization is between 0.797 and 0.799;
- The mean number of tokens in FreePumps is 0.201 ± 0.001; that is, with 90% confidence the mean number of free pumps is between 0.200 and 0.202;
- The average flow time of served cars is 8.945 ± 0.020 minutes; that is, with 90% confidence the average flow time is between 8.925 and 8.965; and
- The fraction of cars rejected is 0.088 ± 0.001; that is, with 90% confidence we can conclude that between 8.7% and 8.9% of cars is not served.

Confidence intervals allow us to compare different alternatives. CPN Tools automatically calculates these intervals when the user uses the command:

```
CPN'Replications.nreplications 10
```

To create more subruns, the parameter of this command can be modified. As discussed in [6], there is a tradeoff between the number of subruns and the length of each run.

7.5 Comparing Alternatives

After creating the CPN shown in Fig. 24 and adding the monitors, it is easy to explore various alternative models and compare them. Table 2 shows the results for the original model (i.e., Fig. 24) and three alternative models.

Table 2. Statistics of the original process and three redesigns (confidence interval of 90%)

Process	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
original	0.900 ± 0.006	0.201 ± 0.001	0.798 ± 0.001	8.945 ± 0.020	0.088 ± 0.001
extra places	1.732 ± 0.014	0.154 ± 0.001	0.846 ± 0.001	12.157 ± 0.050	0.032 ± 0.001
2nd pump	0.107 ± 0.001	1.129 ± 0.002	0.871 ± 0.002	5.431 ± 0.003	0.004 ± 0.001
faster pump	0.389 ± 0.002	0.401 ± 0.001	0.599 ± 0.001	5.541 ± 0.009	0.022 ± 0.001

In the original CPN, MAX_CAPACITY was set to 3, indicating that at most three cars can queue. If there are three cars in the queue and a fourth one arrives, it is rejected (transition Reject fires). Table 2 shows what happens if three more places are added; that is, MAX_CAPACITY is set to 6 meaning that up to six cars can queue. More cars are being served, but waiting times and the average queue length get longer. The flow time increases from approx. 9 minutes to 12 minutes, whereas the percentage of rejected cars decreases to approx. 3%. As the confidence intervals are narrow and non-overlapping, it is justified to make such conclusions.

Instead of adding additional space to wait, we also consider adding an extra pump. This can be done by adding another token to place FreePumps. This change is costly, but has a positive effect on all performance indicators. As Table 2 shows, the flow time drops to approx. 5 minutes and less than 0.5% of cars are rejected.

The last row of Table 2 shows what happens if we replace the original pump by a new one that is 30% faster. This alternative is realized by changing the function that models the service time:

```
fun st(c) = uniform(1.4,3.5);
```

The flow time of this alternative is comparable to adding another pump, but more cars are rejected (approx. 2%).

Each of the three redesigns mentioned in Table 2 can be modeled in less than a minute. This illustrates that simulation using CPN Tools allows for a quick exploration of different alternatives.

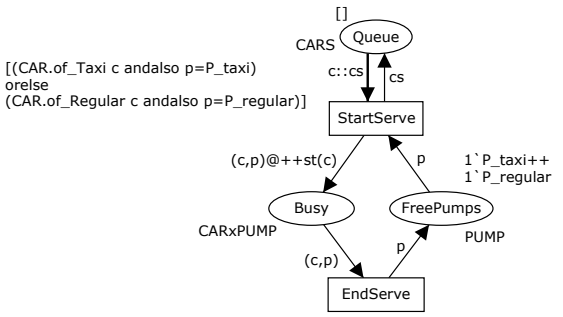


Fig. 25. Changes required to model that taxis have a dedicated pump, i.e., there are two pumps, one for taxis and one for regular cars

To conclude this section, we look at three more redesigns that all distinguish between regular cars and taxis. Figure 25 shows the adaptations needed to add an extra pump such that the new pump is only used by taxis whereas the existing pump is used for regular cars. The type PUMP has now two possible values P_taxi and P_regular and initially there is one of each. The guard of StartServe ensures that the pumps are used for serving the right type of cars.

Table 3 shows the simulation results. The performance is not as good as adding an extra pump that can be used by any type of car. The flow time increases from 5.431 ± 0.003 to 6.800 ± 0.008 and the fraction of rejected cars increases from 0.004 ± 0.001 to 0.033 ± 0.001. There are two reasons for this. First of all, it may be that there are three taxis waiting while the pump for regular cars is free; that is, capacity is unused at times. Second, it may even be that the first car in the row is "blocking" other cars that could be served. For example, consider the scenario with two taxis and two regular cars. If one taxi is being served while the other taxi is first in line, then the two regular cars get blocked even though their pump is idle.

Table 3 shows results for all cars and results for taxis and regular cars separately. Note that the waiting time of taxis is lower, because on average only 5 taxis arrive each hour, whereas on average 10 regular cars arrive each hour.

Next, we consider the situation where taxis have priority; that is, as long as there are taxis waiting, regular cars are not served. This can be realized by simply sorting the queue such that taxis are always in front of the queue. Table 4 shows the results. The overall flow time is comparable to the original situation. However, taxis have, on average, shorter waiting times than regular cars.

The last redesign we consider, reserves the places in the queue for taxis. Taxis can queue as long as there are free slots in the queue. However, regular cars

Table 3. Statistics for the redesign with a dedicated pump for taxis and a dedicated pump for regular cars (confidence interval of 90%)

	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
all cars	0.435 ± 0.002	1.153 ± 0.001	0.847 ± 0.001	6.800 ± 0.008	0.033 ± 0.001
taxis				6.281 ± 0.010	0.033 ± 0.001
regular cars				7.058 ± 0.008	0.033 ± 0.001

Table 4. Statistics for the redesign with a single queue where taxis have priority (confidence interval of 90%)

	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
all cars	0.900 ± 0.004	0.201 ± 0.001	0.798 ± 0.001	8.944 ± 0.014	0.088 ± 0.001
taxis				6.857 ± 0.007	0.088 ± 0.001
regular cars				9.988 ± 0.021	0.088 ± 0.001

can only enter the queue if no other cars are waiting. Hence, regular cars are always rejected unless the queue is empty. Figure 26 shows how the model can be adapted to realize this redesign. The guards of transitions Enqueue and Reject are modified to enforce the new policy. Table 5 shows the results. Compared to the original situation, the fraction of rejected cars increased from 0.088 ± 0.001 to 0.200 ± 0.001; that is, approximately twice as many cars are rejected. However, fewer taxis are rejected (only 0.005 ± 0.001) and the flow time reduced decreased for all cars (both taxis and regular cars).

The last three redesigns show that it is easy to use properties of the car when defining policies. This demonstrates the power of CPNs compared to timed Petri nets that do not incorporate data.

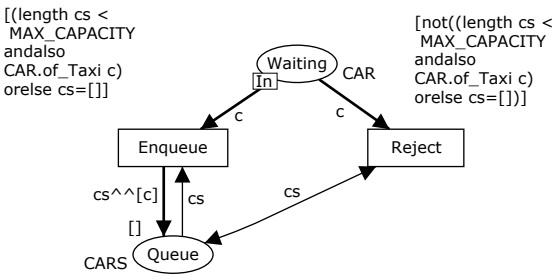


Fig. 26. Only queue if empty

Table 5. Statistics for the redesign in which regular cars can only queue if the queue is empty (confidence interval of 90%)

	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
all cars	0.346 ± 0.001	0.300 ± 0.001	0.700 ± 0.001	6.730 ± 0.007	0.200 ± 0.001
taxis				7.475 ± 0.014	0.005 ± 0.001
regular cars				6.204 ± 0.003	0.297 ± 0.001

8 Conclusion

Colored Petri Nets (CPNs) enhance classical Petri nets with commonly agreed upon extensions such as data, hierarchy, and time. The resulting modeling language is highly expressive and is supported by CPN Tools, a powerful software tool for the modeling and analysis of CPNs.

This paper used a running example to explain several design patterns for modeling in terms of CPNs. These patterns guide users in modeling complex processes that require interplay of control-flow and data-flow. We showed examples of simple patterns and more involved ones.

We also introduced the new features of CPN Tools. Version 3.0 supports priorities and one can now use real values as time stamps. Moreover, the new simulator is up to twice as fast and is now also supported on 64 bit platforms. These improvements facilitate simulating complex processes. Using our running example, we showed that it is easy to generate various alternative designs and using simulation to compare them.

We hope that this paper will help students, researchers, system designers and process analysts to make better models in less time. For a more in-depth discussion on modeling in terms of CPNs, we refer to [6,19]. For the software and examples, we refer to CPN Tools Web page `http://cpntools.org`.

References

1. van der Aalst, W.M.P.: Interval Timed Coloured Petri Nets and their Analysis. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 453–472. Springer, Heidelberg (1993)
2. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Berlin (2011)
3. van der Aalst, W.M.P., de Crom, P.J.N., Goverde, R.R.H.M.J., van Hee, K.M., Hofman, W.J., Reijers, H.A., van der Toorn, R.A.: Ex*Spect* 6.4 An Executable Specification Tool for Hierarchical Colored Petri Nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 455–464. Springer, Heidelberg (2000)
4. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)

5. van der Aalst, W.M.P., Mooij, A.J., Stahl, C., Wolf, K.: Service Interaction: Patterns, Formalization, and Analysis. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 42–88. Springer, Heidelberg (2009)
6. van der Aalst, W.M.P., Stahl, C.: Modeling Business Processes: A Petri Net Oriented Approach. MIT Press, Cambridge (2011)
7. Alexander, C.: A Pattern Language: Towns, Building and Construction. Oxford University Press (1977)
8. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. In: Cambridge Tracts in Theoretical Computer Science, 1st edn. Cambridge University Press (December 2009)
9. Balbo, G.: Introduction to Generalized Stochastic Petri Nets. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 83–131. Springer, Heidelberg (2007)
10. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software through Process Technology. Wiley & Sons (2005)
11. Fokkink, W.: Introduction to Process Algebra. Springer, Berlin (2010)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison Wesley, Reading (1995)
13. Genrich, H.J., Lautenbach, K.: System modelling with high level Petri nets. Theoretical Computer Science 13, 109–136 (1981)
14. Girault, G., Valk, R. (eds.): Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications. Springer, Berlin (2003)
15. van Hee, K.M.: Information System Engineering: a Formal Approach. Cambridge University Press (1994)
16. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley Professional, Reading (2003)
17. Jensen, K.: Coloured Petri Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 248–299. Springer, Heidelberg (1987)
18. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. EATCS monographs on Theoretical Computer Science. Springer, Berlin (1996)
19. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer, Berlin (2009)
20. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer 9(3-4), 213–254 (2007)
21. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An Extensible Editor and Simulation Engine for Petri Nets: Renew. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 484–493. Springer, Heidelberg (2004)
22. Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. Wiley series in parallel computing. Wiley, New York (1995)
23. Merlin, P., Faber, D.J.: Recoverability of Communication Protocols. IEEE Transactions on Communication 24(9), 1036–1043 (1976)
24. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML. The MIT Press (May 1997)
25. Mulyar, N., van der Aalst, W.M.P.: Patterns in Colored Petri Nets. BETA Working Paper Series, WP 139. Eindhoven University of Technology, Eindhoven (2005)

26. Mulyar, N., van der Aalst, W.M.P.: Towards a Pattern Language for Colored Petri Nets. In: Jensen, K. (ed.) Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005), Aarhus, Denmark. DAIMI, vol. 576, pp. 39–48. University of Aarhus (October 2005)
27. Rozinat, A., Mans, R.S., Song, M., van der Aalst, W.M.P.: Discovering Simulation Models. Information Systems 34(3), 305–327 (2009)
28. Rozinat, A., Wynn, M., van der Aalst, W.M.P., ter Hofstede, A.H.M., Fidge, C.: Workflow Simulation for Operational Decision Support. Data and Knowledge Engineering 68(9), 834–850 (2009)
29. Trčka, N., van der Aalst, W.M.P., Sidorova, N.: Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 425–439. Springer, Heidelberg (2009)
30. Valk, R.: Object Petri Nets. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 819–848. Springer, Heidelberg (2004)
31. Weber, B., Reichert, M., Rinderle-Ma, S.: Change Patterns and Change Support Features: Enhancing Flexibility in Process-Aware Information Systems. Data and Knowledge Engineering 66(3), 438–466 (2008)
32. Workflow Patterns Home Page, `http://www.workflowpatterns.com`

Appendix: CPN Patterns

In this paper, we used and described several CPN patterns without explicitly enumerating them. This way we could focus on the running example and the process of modeling in terms of CPNs. In this appendix, we list the patterns described in [25,26]. This was the first systematic collection of design patterns for CPNs. Note that this collection was based on constructs described in books such as [6,15,18,19] and examples on the CPN Tools website.

1. *ID Matching*: to make identical information objects distinguishable. This pattern is used in most CPN models shown in this paper. For example, individual cars are distinguished using type CAR.
2. *ID Manager*: to ensure uniqueness of identifiers used for distinguishing identical objects. The places labeled Next Car ID in Figs. 1, 7(left), 19(right), 21, and 24(right) are used to realize this pattern, i.e., each car gets a unique ID.
3. *Aggregate Objects*: to allow manipulation of a set of information objects as a single entity. The queue pattern is a specialization of this pattern and used in various versions of the gas station module (see below). Place Receivers in Fig. 16 is another example implementing this pattern.
4. *Queue*: to allow manipulation of the queued objects in a strictly specified order. The places labeled Queue in Figs. 12(left), 18(left), 19(left), 20, and 24(left) are used to realize this pattern, i.e., cars are put into a list to enforce a particular order.
5. *FIFO Queue*: to allow manipulation of objects from the collection in a strictly specified order such that an object which arrived first is consumed first. The places labeled Queue in Figs. 12(left), 18(left), 19(left), 20, and 24(left) are used to realize this pattern, i.e., cars are put into a list and when a pump becomes available the car that queued first is taken.
6. *LIFO Queue*: to allow manipulation of objects from the collection in a strictly specified order, such that the mostly recently added object is retrieved first. Most queues in this paper use a FIFO order. By taking the last element from the list rather than the first, a FIFO queue can be converted into a LIFO queue.
7. *Random Queue*: to allow manipulation of objects from the collection such that objects are added to the queue in any order, and an arbitrary object is consumed from it.
8. *Priority Queue*: to allow manipulation of objects from the collection in the order of the objects' priority. Table 4 shows simulation results for a CPN model using this pattern.
9. *Capacity Bounding*: to prevent over-accumulation of objects in a certain place. This was the main topic of Sect. 4.1. See place Being Served in Fig. 9 (middle) and place Queue in Fig. 9(right). Both places are bounded by a complement place. The pattern is also used in Figs. 10, 11, 12, 13, 14, 17, 18(left), 19(left), 20, 24(left), and 25.

10. *Inhibitor Arc*: to support "zero-testing" of places. Transition Reject in Fig. 10(left) can only fire if place Queue contains two tokens and, hence, all free places are taken. Fig. 10(right) models this more explicitly by adding a place that counts the number of free places. In Fig. 26, regular cars can only queue if no taxis are waiting.
11. *Colored Inhibitor Arc*: to support "non-containment" property of places.
12. *Shared Database*: to enable centralized storage of data shared between multiple transitions, supporting different levels of data visibility (i.e. local, group, or global).
13. *Database Management*: to specify the interface of accessing data, stored in a shared database for read-only and modification purposes.
14. *Copy Manager*: to make data stored in the shared database available at other locations for local use, while maintaining the consistency of data in all places.
15. *Lock Manager*: to synchronize access to shared data by means of exclusive locks. The fuel stations are shared resources that can only be used by one car at a time. In Fig. 25 taxis and regular cars have a dedicated pump. This can be seen as a form of locking.
16. *Bi-Lock Manager*: to synchronize access to shared data for reading and writing purposes by means of shared and exclusive locks.
17. *Log Manager*: to record the information about actual process execution by means of a data log.
18. *Blocking State-Independent Filter*: to prevent data non-conforming to a certain property from passing through.
19. *Blocking State-Dependent Filter*: to prevent data non-conforming to a property involving the state of an external data-structure, from passing through. Transition Reject in many of the variants of the gas station module is only allowing cars to queue if the queue is not full yet. Consider for example the two variants shown in Fig. 10. Taxis that cannot enter the queue will not be rejected; they are blocked until the queue is not fully occupied anymore. This can be seen as a variant of this pattern.
20. *Non-Blocking State-Independent Filter*: to filter-out data fulfilling a certain property while avoiding accumulation of non-conforming data in the filter input place.
21. *Non-Blocking State-Dependent Filter*: to filter-out data non-conforming to a property, involving the state of an external data-structure, while avoiding accumulation of non-conforming data in the filter input. Consider for example the two variants shown in Fig. 10. Regular cars that cannot enter the queue are immediately rejected; as long as the queue is fully occupied, regular cars are filtered out. This can be seen as a variant of this pattern. Another example is Fig. 26 where regular cars can only queue if no taxis are waiting.
22. *Translator*: to enable coordinated communication between two actors with originally different data formats.
23. *Asynchronous Transfer*: to allow transportation of data from one location to another, while avoiding the sender to block. The environment module in Fig. 6 does not block while sending cars to the gas station module.

24. *Synchronous Transfer*: to allow transportation of data from one location to another, ensuring that an actor, which posted a request, is blocked until it does not receive the requested information.
25. *Rendezvous*: allow multiple actors to broadcast and discover data objects concurrently.
26. *Asynchronous Router*: to enable asynchronous transfer of data from a single source to a dedicated target, providing loose coupling between the source and targets connected to it.
27. *Asynchronous Aggregator*: to provide a holistic view of data, produced by multiple unrelated sources through asynchronous data aggregation.
28. *Broadcasting*: to allow broadcasting of data from a single source to multiple targets, while avoiding direct dependencies between them. Section 5.2 shows how to realize a broadcast (see for instance Fig. 16).
29. *Redundancy Manager*: to prevent the transfer of duplicated data between loosely-coupled actors who communicate asynchronously.
30. *Data Distributor*: to support parallel data processing by distributing data between several independent actors.
31. *Data Merger*: to compose a single information object out of several smaller ones when all parts required for composition become available.
32. *Deterministic XOR-Split*: to allow at most one transition out of several possible to execute, based on fulfillment of mutually excluding data conditions. Transitions Enqueue and Reject in Fig. 24(left) form a deterministic XOR-split.
33. *Non-Deterministic XOR-Split*: to allow any transition out of several possible, but satisfying the same data condition, to execute. Transitions Serve and Reject in Fig. 1 form a (partially) non-deterministic XOR-split; regular cars can be served or rejected in a non-deterministic manner.
34. *OR*: to allow any number of tasks to be selected for execution based on the fulfillment of a certain data condition.

Some of the patterns just mentioned (e.g., the *Queue* and *Capacity-bounding* patterns) are used frequently in this paper, others are not used at all. The 34 patterns described in [25,26] focus on data management and communication. This explains why several patterns are not used in our running example. Some of the patterns presented in this paper are closer to the workflow patterns mentioned in Sect. 1 [4,32] rather than the original CPN patterns. For example, the *Region Flush* pattern discussed in Sect. 5.3 is closely related to the *Cancel Region* pattern in [4,32]. Moreover, compared to the original CPN patterns in [25,26], we put more emphasis on time (see Sect. 6 and Sect. 7).

Applications of Coloured Petri Nets for Functional Validation of Protocol Designs

Lars M. Kristensen[1,⋆] and Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computer Engineering, Bergen University College, Norway
{lmkr,kifs}@hib.no
[2] DTU Informatics, Technical University of Denmark, Denmark
kisi@imm.dtu.dk

Abstract. Communication protocols constitute central building blocks in most modern IT systems as they define components, rules, and languages that make data communication possible. The development of correct protocols is a challenging engineering discipline, making modelling and validation of protocol design an important application domain for Coloured Petri Nets (CPNs). We illustrate the practical application of CPNs for protocol validation by focusing on selected aspects of four recent projects involving industrial-sized protocols. These projects demonstrate how CPNs can be used to model protocol elements and improve protocol specifications, how state space exploration can be used to verify protocol properties, and how behavioural visualisation in combination with a CPN model provides an effective way of rapidly constructing an executable prototype of a protocol design.

1 Introduction

Communication protocols play an important role in most IT systems. A prominent example is the vast amount of web applications that are in use today for, e.g., online banking, shopping, government administration, and entertainment. The services provided by these applications all rely on the protocols governing the operation of the Internet. Other examples are telecommunication systems, logistic systems with sensors and actuators, and control systems in vehicles. All these systems rely heavily on communication and synchronisation between concurrently executing software components and subsystems. As protocols are to support still more complex services that are critical to both the operation of companies and the everyday life of citizens, it is important that they are working correctly already from the initial deployment.

Protocol engineering [80] typically involves a specification of the *service* that the protocol is to provide. Through a synthesis or design step, a protocol design is developed with the aim of providing the desired service. For protocol design, *functional* and *performance* validation can be conducted to investigate and reason about the properties of the design. Functional validation focuses on the

⋆ Work supported by the Research Council of Norway project 194521 (FORMGRID).

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 56–115, 2013.

logical correctness of the protocol such as the absence of deadlocks and livelocks, that a request is always followed by a response, or whether the proposed protocol design provides the desired services. Performance validation is concerned with quantitative properties such as delays, throughput, and response time. Eventually, the protocol design is implemented and may then be subject to further testing.

Protocol design is in many cases a challenging task. One reason for this is that the execution of a protocol can proceed in different ways, e.g., depending on which messages are lost in transmission, the scheduling of the protocol entities, the time at which events are received from the environment of the protocol, and the execution path taken by the protocol entities. Another reason is that a protocol by nature involves independently scheduled entities which makes testing and reproduction of executions difficult. All this means that protocols often have a very large number of possible executions. In this process, it is easy for a protocol engineer to overlook important interaction patterns which may in turn lead to gaps or malfunction of the protocol.

The specification of the protocol service and the protocol design is, in many cases, based on natural language descriptions. One example of this is the Request for Comments (RFC) documents published by the Internet Engineering Task Force (IETF) [47]. Natural language specifications of protocols often have many issues that needs to be resolved before a properly working implementation can be obtained. One class of issues originates from the fact that such specifications are inherently ambiguous making it difficult to achieve inter-operability between independent implementations. Another source of issues to resolve is that the specifications are often incomplete in that the behaviour of the protocol is not described for all cases.

The challenges outlined above have made protocols a prominent application domain for formal description techniques [46], including Petri Nets [93, 97]. In this paper we concentrate on the use of Coloured Petri Nets (CPNs) [56, 59, 61] for modelling and functional validation of protocol designs. Our purpose is to provide an introduction to, and an overview of, how CPNs have been applied for practical validation of protocol designs. We approach this by presenting selected parts of CPN models and associated results originating from projects conducted in an industrial context with industrial-sized protocols. More specifically, we present in the core of this paper the application of the CPN modelling language, tools, and techniques for functional validation of the following protocols:

The DYMO Routing Protocol. The Dynamic On-Demand Routing Protocol for Mobile Ad-hoc Networks (DYMO) [15] is a routing protocol for mobile ad-hoc networks being developed by the MANET working group of the IETF. The DYMO case study is used to illustrate *protocol modelling* with CPNs and to introduce the basic constructs of the CPN modelling language. Our presentation is based on the CPN model constructed in a project on modelling and validating DYMO [25].

The Generic Access Networks (GAN) Architecture. The GAN protocol architecture [2] is developed by the 3rd Generation Partnership Project (3GPP)

for accessing telephone services via Internet Protocol (IP) networks. The GAN case study is used to introduce the basics of *explicit state space exploration* and show how it can be used in a fully automatic manner as a first step in the *verification* of a protocol design. The presentation is based on the project [30] conducted at TietoEnator A/S to specify the detailed usage of protocol software and services via specialisation of the GAN protocol architecture.

The Routing Interoperability Protocol (RIP). The RIP protocol developed at Ericsson Telebit A/S enables routing of IP packets between core IP networks and mobile ad-hoc networks. The RIP case study is used to illustrate how *application-specific behavioural visualisation* can be applied on top of CPN models. In particular, how it can be used to obtain a first executable prototype of the protocol design allowing for early experiments and for presentation to customers and management with the aim of soliciting protocol design requirements. Our presentation is based on the project [74] conducted in cooperation with Ericsson where CPN modelling was used to specify the operation of the RIP protocol.

The Edge Router Discovery Protocol (ERDP). The ERDP protocol is an IPv6-based protocol allowing edge routers to configure gateways in mobile ad-hoc networks with IP address prefixes. The ERDP case study is used to illustrate how the combined use of CPN modelling, state space exploration, and behavioural visualisation all contributed to identify and resolve design issues and errors during ERDP development. Our presentation is based on the project [67] conducted at Ericsson Telebit A/S on the design of the ERDP protocol.

The rest of this paper is organised as follows. Section 2 provides a high-level overview of CPNs and related techniques used for functional validation of protocol designs. Sections 3-6 then present the application of CPNs on the four protocols introduced above. In Sect. 7 we survey related work where CPNs have been used for protocol validation. Finally, Sect. 8 contains conclusions and outlines directions for future work. The reader is assumed to be familiar with the basic ideas of Petri nets [97] and TCP/IP communication protocols [21]. The reader is referred to [61] for a comprehensive introduction to CPNs, state space exploration, and behavioural visualisation of CPN models.

2 Background: CPNs and Functional Protocol Validation

The CPN modelling language belongs to the family of High-level Petri Nets and combine Petri Nets with the Standard ML (SML) programming language [100]. Petri Nets provide the foundation of the graphical notation and the semantical foundation for modelling concurrency, synchronisation, and communication. The functional programming language SML provides primitives for representing sequential aspects of protocols (such as data manipulation) and for creating compact and parameterisable models. Formal modelling and validation with CPNs

is supported by CPN Tools [95] which provides support for construction, simulation, functional and simulation-based performance analysis of CPN models. The addition of data types and a high-level programming language offered by CPN (in contrast to ordinary Petri nets) is highly important when constructing Petri net models of protocols. As an example, with ordinary Petri nets each message type exchanged between protocol entities need to be present with multiple places, and data manipulation (e.g., comparison of data packet content such as sequence numbers) needs to be modelled relying only on net structure resulting in models that are difficult to comprehend.

The advantage of CPNs (and formal description techniques in general) is that they are based on the construction of executable models that make it possible to observe and experiment with the protocol design prior to implementation and deployment using, e.g., simulation. This typically leads to more complete protocol specifications since the model will not be fully operational until all parts of the protocol have been (at least abstractly) specified. Furthermore, the construction of a formal and executable model helps identify and resolve ambiguities that may be present in a natural language specification. Another advantage is the support for model abstractions that makes it possible to specify the operation of the protocol without being concerned with implementation details such as message layout. A model also makes it possible to explore larger scenarios of a protocol system than what is in many cases practically possible in a laboratory.

2.1 Simulation and Behavioural Visualisation

During a protocol model construction phase it is common to use *interactive simulation* of the CPN protocol model to investigate the operation of the protocol in detail. An interactive simulation is similar to single-step debugging and the execution of the CPN model is viewed directly on its graphical representation and provides a simple way of validating that the model operates as intended. In an interactive simulation, the modeller is in charge and determines the next step by selecting between the enabled events in the current state. Interactive simulation is typically combined with the use of *automatic simulation* which is similar to program execution and the purpose is to execute the CPN model without detailed interaction and inspection. Automatic simulation is typically used for testing purposes, and the modeller typically sets up appropriate breakpoints and stop criteria.

Even though the CPN modelling language supports abstraction and hierarchical modules there can still be a significant amount of detail being presented with this approach, and observing every single step either in an interactive simulation or in a log file based on an automatic simulation is often too detailed a level of observation when investigating the behaviour of a model. Furthermore, even if the CPN model is executable, it still lacks the application- and domain-specific appeal of a conventional software prototype. CPN Tools can use the BRITNeY Suite animation framework [111] to create behavioural visualisation [112] and interaction graphics on top of CPN models. The animation framework is a stand-alone application, and CPN Tools invokes the primitives of the animation

framework using remote procedure calls. The animation framework supports a wide range of diagram types via a plug-in architecture that makes it possible to visualise the execution of protocols using both standard diagrams (e.g., message sequence charts) in addition to tailored, application-specific diagrams. In this way it is possible to investigate the behaviour of a protocol design while overcoming the limitations of interactive and automatic simulations. In this paper we give some examples of both standard and application-specific diagrams. The reader is referred to [111] for a comprehensive introduction to the animation framework.

2.2 State Spaces and Verification

Verification of behavioural properties of protocols with CPNs [66] is supported by explicit *state space exploration* [6]. In its simplest form this approach involves computing a directed graph where the nodes corresponds to the set of reachable states of the CPN model and the arcs represent occurrences of events causing state changes. State spaces can be constructed fully automatically by the state space tool in CPN Tools and guarantees complete coverage of all executions. State space hence provides a highly systematic error-detection technique that make it possible to automatically (i.e., algorithmically) check whether a protocol has a formally stated desired property. In addition, state space methods have the advantage that counter examples (error-traces) can be automatically synthesised if the protocol does not satisfy a given property.

The main disadvantage of state space exploration is the inherent state explosion problem [103], and a multitude of advanced state space methods have been developed aimed at alleviating the inherent state explosion problem. Early work on addressing state explosion in the context of CPNs concentrated on computer tool support for, and initial experiments with, the equivalence [57], symmetry [20, 24, 48, 58], and the stubborn set methods [102]. The symmetry and equivalence methods rely on constructing a condensed state space where each node represents an equivalence class of states and each arc represents an equivalence class of events. The symmetry method has, e.g., been applied on a mutual exclusion protocol [62] and an embedded systems protocol [81]. The equivalence method has only been used on a small stop-and-wait protocol [63] due to the obligation of providing a manual soundness proof for the user-provided equivalence relation. The stubborn set method [101, 103] relies on analysing enabling and disabling dependencies between events and use this to explore only a subset of the events in each state encountered during state space exploration. The rich SML-based inscription language which is fundamental building block of the CPN modelling language, however, poses problems for the analysis of transition dependencies in the context of CPNs [72] – unless relying on an unfolding of the CPN model to the equivalent Place/Transition net. Hence, restrictions on the modelling language are required to apply the stubborn set method without relying on unfolding. Another widely used verification approach in the context of CPNs is based is the methodology of [9]. A central component of this approach is an explicit modelling of both the protocol and its service, and the use

of finite-state automata language comparison as a criteria for checking that the protocol conforms to the specified service. Recent work on addressing the state explosion problem in the context of CPNs has concentrated on making more economical use of memory resources when exploring the state space. Memory is (in many cases) the limiting factor in state space exploration of CPN models due to the large state vectors. This work resulted in the development of the sweep-line method [19, 60] and the comback method [27, 110]. The sweep-line suite of methods [8, 19, 68, 69, 83] is aimed at on-the-fly verification and exploits a notion of progress found in many concurrent systems. Exploiting progress allows for the deletion of states from memory during a progress-first traversal of the state space. This in turn reduces peak memory usage. The sweep-line method has been used [34, 35, 41, 105] for the verification of several industrial-sized protocols specified using the CPN modelling language. The comback method can be viewed as an exploration-order independent storage mechanism based on hash compaction [98, 113]. It allows the usually large state vectors of CPN models to be stored in compact form, and the full state vector of a state is reconstructed when needed for comparison with newly generated states. Unlike the classical hash compaction method, the comback method guarantees full coverage of the state space. The ASAP model checking platform [109] has support for a number of these advanced state space methods – including methods developed outside the context of CPNs.

2.3 Formal Specification Techniques for Protocols

CPNs and Petri Nets represents one approach to the formal specification and verification of protocols. Historically, several non-Petri nets based languages targeting protocol specification have been developed, in particular in relation to telecommunication standardisation efforts [75, 94]. The Language of Temporal Ordering Specification (LOTOS) [1, 14, 50] was developed as part of International Standardisation Organisation (ISO) efforts and linked to the development of the Open Systems Interconnection (OSI) reference model. LOTOS is founded on the Calculus of Communicating (CCS) [86] and add a data type component to CCS based on algebraic specification. The Extended State Transition Language (Estelle) [49] also originated from OSI standardisation efforts and is based on extended finite state machines [13] combined with extensions to the PASCAL programming language. The Specification and Description Language (SDL) [55] has evolved in several generations since 1980 within the International Telecommunication Union - Telecommunication Sector (ITU-T). SDL is based on communicating extended state machines and has in later versions been equipped with a formal semantics [55] making it amendable for formal verification. A Unified Modelling Language (UML) Profile [52] linking SDL and UML also exists. A comparison of these classical specification languages can be found in [5]. Estelle, SDL, and CPNs are all equipped with a language for modelling data manipulation, but have a different theoretical foundations (extended state machines versus Petri Nets). Another difference is that CPNs have very few (but still powerful) modelling constructs in contrast to languages such as Estelle and

SDL which have a large and complex set of language constructs to describe the behaviour of protocol entities and their interaction. From this perspective, CPNs provide a simpler and more lightweight approach to protocol modelling which at the same time less implementation specific than, e.g., typical SDL protocol specifications. In that respect, CPNs are close to languages like LOTOS that focus more on abstract and implementation independent protocol specification. Within ITU-T languages has also been developed related to protocol data representation. The Abstract Syntax Notation One (ASN.1) [53] is a notation of describing data structures carried in messages exchanges between protocol entities. The Encoding Control Notation (ECN) [54] is a language for specifying ASN.1 encoding rules. In terms of specification of data structures, the SML data types for defining colour sets in CPNs provide similar capabilities as ASN.1. The Testing and Test Control Notation 3 (TTCN-3) [26] is a language for writing protocol test specification.

The Process Meta Language (Promela) language [46] providing the modelling foundation of the SPIN tool [45] has been widely used for protocol design and verification. Promela is based on Communication Sequential Processes (CSP) [44] and is in contrast to CPNs, a textual modelling language with a different theoretical foundation. In a UML context, state diagrams (charts) [43] are used for modelling protocol modules (e.g., [84]), and message sequence charts (MSCs) [51] (sequence diagrams in UML) are being used in particular for specifying protocols requirements that can later be used in protocol verification [4, 38]. MSCs have also been used for protocol specification using higher-level control flow constructs. In contrast to MSCs which are action-oriented, then state charts and CPNs are both state and action-oriented modelling formalisms. Timed automata [7] as supported, e.g., by the UppAAl tool has also been used for the specification and verification of protocols (e.g., [29, 96]). The UppAal models consists of a network of network of communicating timed automata, and are specifically suited for modelling and verifying protocol where continuous timing constraints are essential. In comparison, the timed concepts provided by CPNs is a discrete time concept of time. An example on the use of CPNs to model protocols with time constraints can be found in [71].

3 The DYMO Protocol

Modelling a protocol involves developing a representation of the *messages* (or packets) exchanged between the *protocol entities*, the *procedure rules* and *internal state* of the protocol entities guarding the processing of messages, and developing a model of the *environment* in which the protocol is being executed. The environment model typically encompass an abstract representation of the communication medium (or channel) over which the protocol operates. The primary purpose of this section is to illustrate how these protocol elements can be represented in the CPN modelling language using the Dynamic On-demand Routing Protocol (DYMO) [15] for mobile ad-hoc networks as an example. This section additionally shows how to construct compact parameterised CPN models

where the number of protocol entities can easily be configured, and how communication networks with a dynamic topology can be modelled.

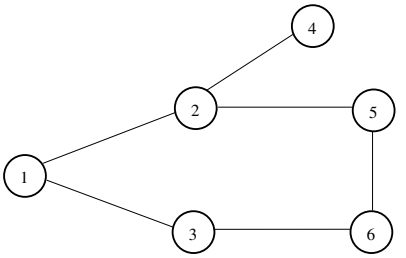
3.1 MANETs and Operation of the DYMO Protocol

A *mobile ad-hoc network* (MANET) comprises a collection of mobile nodes, such as laptops, personal digital assistants, and mobile phones, capable of establishing a communication infrastructure for their common use. Ad-hoc networking differs from conventional networks in that the nodes operate in a fully self-configuring, autonomous and distributed manner, without any preexisting communication infrastructure such as base stations and routers. Network layer and routing protocols for ad-hoc networking (including the DYMO protocol) are currently under development by the IETF MANET working group.

The operation of the DYMO protocol consists of two parts: *route discovery* and *route maintenance*. Route discovery is used to establish routes between nodes and begins with an *originator node* multi-casting a Route Request (RREQ) message to all nodes in its immediate range. A RREQ message has a *sequence number* to enable other nodes in the network to judge the freshness of the route request. The ad-hoc network is then flooded with RREQs until the request reaches the *target node* (provided that there exists a path from the originating node to the target node). The target node replies with a Route Reply (RREP) message unicasted hop-by-hop back to the originator node. The route discovery procedure is requested by the Internet Protocol (IP) layer on a node when it receives an IP packet for transmission and does not have a route in its routing table to the target node.

Figure 1(left) depicts the topology of a MANET consisting of six nodes numbered 1–6. An edge between two nodes indicates that the nodes are within direct transmission range. In this case, we assume that all communication links are symmetric. Figure 1(right) (to be discussed below) lists for each node the *routing table* entries created as a result of executing a routing discovery procedure with node 1 as the originator node and node 6 as the target node. The routing table entries in Fig. 1(right) are specified as a pair (*target*, *nexthop*). The second column specifies the entries that are created as a result of a node receiving the RREQ. The third column lists the entries created as a result of receiving the corresponding RREP. When explaining the operation of the DYMO CPN model below, we will use the scenario in Fig. 1 as a running example.

The *message sequence chart* (MSC) in Fig. 2 depicts one possible exchange of messages in the DYMO protocol when the originating node 1 establishes a route to target node 6 in the topology in Fig 1(left). Solid arcs represent multi-cast transmission and dashed arcs represent unicast transmission. In the MSC, node 1 multi-casts a RREQ which is received by nodes 2 and 3. When receiving the RREQ from node 1, nodes 2 and 3 create an entry in their routing table specifying a route back to the originator node 1. Since nodes 2 and 3 are not the target of the RREQ they both multi-cast the received RREQ to their neighbours (nodes 1, 4 and 5, and nodes 1 and 6, respectively). Node 1 discards these messages as it was the originator of the RREQ. When nodes 4 and 5 receive



Node	RREQ	RREP
1		(6,3)
2	(1,1)	
3	(1,1)	(6,6)
4	(1,2)	
5	(1,2)	
6	(1,3)	

Fig. 1. Example MANET topology (left) and routing table entries (right)

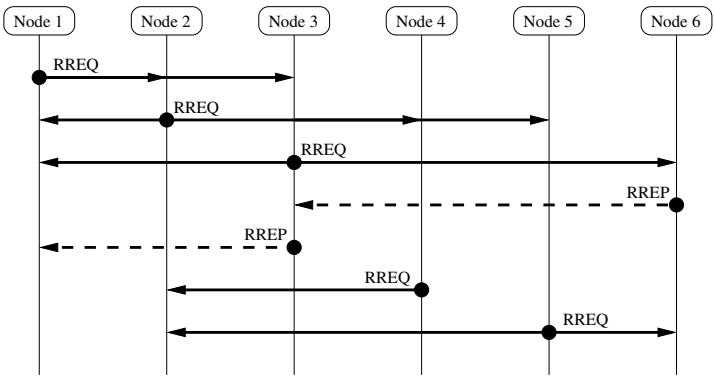


Fig. 2. Message exchange scenario showing DYMO route discovery procedure

the RREQ they add an entry to their routing table specifying that the originator node 1 can be reached via node 2. When node 6 receives the RREQ from node 3, it discovers that it is the target node of the RREQ, adds an entry to its routing table specifying that node 1 can be reached via node 3, and unicasts a RREP back to node 3. When node 3 receives the RREP it adds an entry to its routing table stating that node 6 is within direct range, and use its entry in the routing table that was created when the RREQ was received to unicast the RREP to node 1. Upon receiving the RREP from node 3, node 1 adds an entry to its routing table specifying that node 6 can be reached using node 3 as the next hop. The RREQ is also multi-casted by node 4, but when node 2 receives it again, it will be discarded by node 2 because it has already processed the RREQ message once. Node 5 also multi-casts the RREQ, but nodes 2 and 6 also discard the RREQ message as it has already been received once. From Fig. 1(right) it can be seen that upon completion of the route discovery procedure, a bidirectional route has been discovered and established between node 1 and node 6 using node 3 as an intermediate hop.

The topology of a MANET changes over time because of the mobility of the nodes. DYMO nodes therefore perform *route maintenance* where each node

monitors the links to the nodes it is directly connected to. The DYMO protocol has a mechanism to notify nodes about routes that become broken due to nodes moving out of range of each other. This is done by sending Route Error (RERR) messages which have the effect of informing nodes using the broken route that a new route discovery is needed in order to reestablish a communication path.

3.2 CPN Model Overview and Message Modelling

The DYMO CPN model is a hierarchical model organised in 14 *modules*. Figure 3 shows the *module hierarchy* of the CPN model. Each node in Fig. 3 corresponds to a *module* with System representing the top-level module of the CPN model. An arc leading from one module to another indicates that the latter is a *submodule* of the former. The model is organised into two main parts. The DYMOProtocol module and its nine submodules model the DYMO protocol entities including the internal state of the protocol entities and the procedure rules for receiving messages, internal processing, and sending of messages. The MobileWirelessNetwork module and its two submodules model the environment for the DYMO protocol. This includes the modelling of how messages are transmitted over a wireless link and the modelling of how the mobility of the nodes affects the current topology of the network. The division of the model into submodules reflects the structure of the DYMO specification [16] and hence maintains a close structural relationship between the natural language specification and the formal CPN model. The CPN model does not capture the transmission of payload from the application layer as the focus of the model is on the route establishment and maintenance of the DYMO protocol.

The top-level module System is shown in Fig. 4 and is used to connect the two main parts of the model. It corresponds to the System node in Fig. 3. The

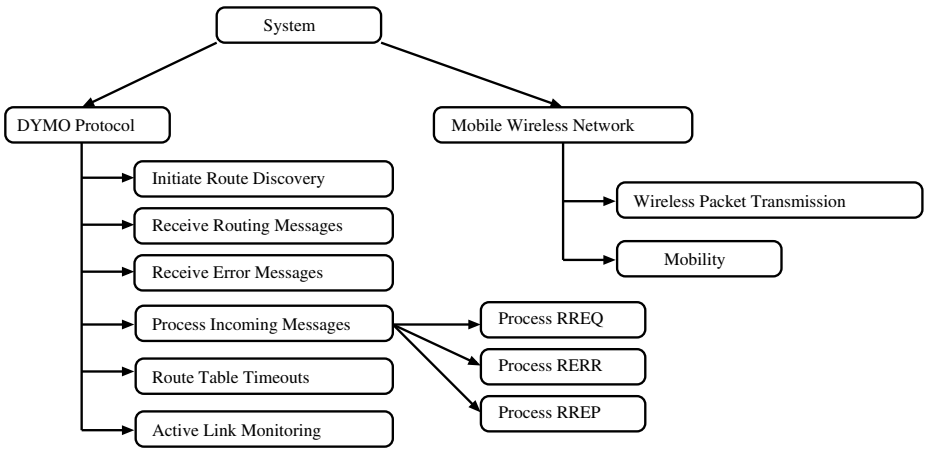


Fig. 3. Module hierarchy for the DYMO CPN model

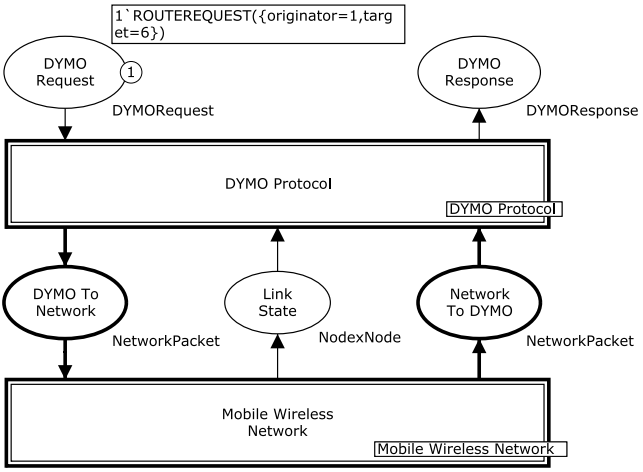


Fig. 4. Top-level System module of the DYMO CPN model

module has two *substitution transitions* drawn as rectangles with double-line borders. Each of the substitution transitions have an associated tag positioned next to it specifying the name of the associated submodule. The DYMOProtocol substitution transition has the DYMOProtocol module as its associated submodule, and the MobileWirelessNetwork substitution transition has the module MobileWirelessNetwork as its associated submodule. In this model, the substitution transition has the same name as its associated submodule (but this is not generally required).

The two *socket places* DYMOToNetwork and NetworkToDYMO connected to the substitution transition DYMOProtocol are used to model the interaction between the DYMO protocol and the MANET environment as represented by the submodules of the MobileWirelessNetwork substitution transition. The socket place LinkState is used to model the *active link monitoring* that nodes perform to check which neighbour nodes are still reachable. When the DYMO protocol module sends a message, it will appear as a token representing a network packet on the socket place DYMOToNetwork. Similarly, a network packet to be received by the DYMO protocol module will appear as a token on the NetworkToDYMO socket place. Each of the socket places in Fig. 4 (places connected to a substitution transition) is associated with a *port place* in the submodule associated with the substitution transition that the socket place is connected to. The association between a socket and a port place has the effect that the port and the socket places will always have identical markings (tokens). An arc leading to a socket place from a substitution transition means that transitions on the submodule associated with the substitution transitions will add tokens on this place. Analogously, an arc leading from a socket place to a substitution transition means that transitions on the submodule will remove tokens from this place.

The colour set (data types) of each place determining the kind of tokens that can reside on the place is written below each place. The colour set declarations used in Fig. 4 is provided in Fig. 5. A record colour set is used for representing the packets transmitted over the wireless links. A `NetworkPacket` consists of a source (field `src`), a destination (field `dest`), and some `data` (payload). The DYMO messages are designed to be carried in User Datagram Protocol (UDP) datagrams. This means that the network packets are abstract representations of IP/UDP datagrams. The model abstracts from all fields in the IP and UDP datagrams (except source and destination fields) as only these impact the DYMO protocol logic. The source and destination of a network packet are modelled by the `IPAddr` colour set. There are two kinds of IP addresses in the model: `UNICAST` addresses and the `LL_MANET_ROUTERS` multi-cast address. The multi-cast address is used, e.g., in route discovery when a node is sending a RREQ to all its neighbouring nodes. Unicast addresses are used as source of network packets and, e.g., as destinations in RREP messages. A unicast address is represented as an integer from the colour set `Node`. Hence, the model abstracts from real IP addresses and identify nodes (communication interfaces) using integers in the interval $[1; N]$ where `N` is a model parameter specifying the number of nodes in the MANET.

```
(* --- Nodes and abstract IP/UDP messages --- *)
colset Node          = int with 0 .. N;
colset IPAddr        = union UNICAST : Node + LL_MANET_ROUTERS;

colset NetworkPacket = record src  : IPAddr * dest : IPAddr *
                              data : DYMOMessage;

(* --- DYMO service --- *)
colset RouteRequest = record originator : Node * target : Node;

colset DYMORequest  = union ROUTEREQUEST : RouteRequest;

colset RouteResponse = record originator : Node * target : Node *
                              status     : BOOL;

colset DYMOResponse  = union ROUTERESPONSE : RouteResponse;
```

Fig. 5. Colour set declarations for nodes, network packets, and DYMO service

The two places DYMORequest and DYMOResponse in Fig. 4 are used to interact with the service provided by the DYMO protocol. A route discovery for a specific destination is requested by putting a token on the DYMORequest place and a DYMO response to a route discovery request is then provided by DYMO

as a token via the DYMOResponse place. The colour sets `DYMORequest` (see Fig. 5) specifies the identity of the `originator` node requesting the route and the identity of the `target` node to which a route is to be discovered. Similarly, a `DYMOResponse` message contains a specification of the `originator`, the `target`, and a boolean `status` specifying whether the route discovery was successful. The colour sets `DYMORequest` and `DYMOResponse` are defined as union types to make it easy to later extend the model with additional requests and responses. By setting the *initial marking* of the place DYMORequest, it can be controlled which route discovery requests are to be made.

The small circles and associated boxes in Fig. 4 show the *current marking* of the CPN model. The small circle positioned inside a place indicates the number of tokens on the place in the current marking. In Fig. 4, there is a single token on the place DYMORequest with colour `ROUTEREQUEST({originator=1,target=6})` as specified in the box positioned next to the small circle. This marking corresponds to the DYMO protocol being requested to establish a route from node 1 to node 6 as considered in the scenario in Fig. 1.

3.3 Modelling the DYMO Protocol Entities

The top-level module for the DYMO protocol part of the CPN model is the DYMOProtocol module shown in Fig. 6. The module has five substitution transitions modelling initiating route requests (substitution transition InitiateRouteDiscovery), reception of RREQ and RREP messages (substitution transition ReceiveRoutingMessages), the reception of RERRs (substitution transition ReceiveErrorMessages), processing of incoming messages (substitution transition ProcessIncomingMessages), and timer management associated with the routing table entries (substitution transition RouteTableEntryTimeouts). The places DYMORequest, LinkState, and NetworkToDYMO are *input port places* of the module as indicated by the In tag positioned next to them. Each of these places are associated with the accordingly named socket places in Fig. 4. Similarly, the places DYMOToNetwork and DYMOResponse are *output port places* as indicated by the Out tag positioned next to them, and they are associated to the accordingly named socket places in Fig. 4.

All submodules of the substitution transitions in Fig. 6 create and manipulate DYMO messages which are represented by the colour sets defined in Fig. 7. The definition of the colour sets used for modelling the DYMO messages is based on a direct translation of the description of DYMO messages as found in the DYMO specification [16]. In particular, the same names of message fields as in [16] have been used. The model abstracts from the compact packet layout defined for the DYMO protocol. This is done to ease the readability of the CPN model, and since the packet layout is not important when considering only the functional operation of the DYMO protocol.

The place RoutingTable and the place OwnSeqNum are used to model the routing table and the sequence number of nodes, respectively, that are maintained as part of the internal state of each mobile node. In the marking depicted in Fig. 6 both of these places contain a multi-set containing six tokens. Within the

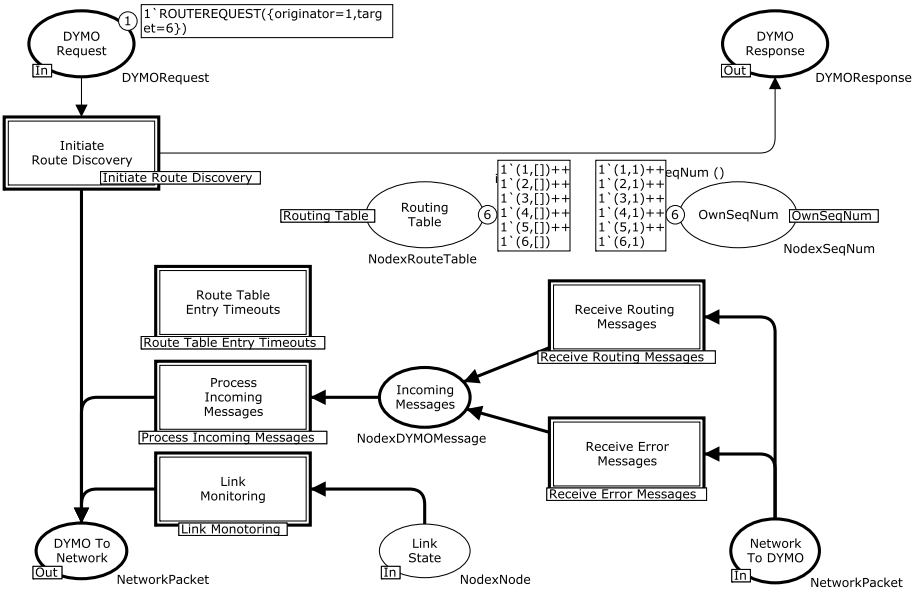


Fig. 6. The DYMOProtocol module

boxes specifying the colours of the individual tokens, ++ (pronounced and) is used to denote union of multi-sets and ‘ (pronounced of) is used to specify the coefficients (i.e., the number of occurrences of tokens with a given colour). The colour set `SeqNum` used to represent the sequence number of a node was defined above, and the colour set `RouteTable` is defined in Fig. 8. To allow each node to have its own sequence number, we use the colour set `NodexSeqNum`. The marking in Fig. 6 corresponds to a MANET with six mobile nodes. The first component of each token on the place OwnSeqNum specifies the identity of a node and the second component specifies the sequence number of the node. Initially, the sequence number of all nodes is set to one. Similarly, it can be seen that the routing table of each mobile node is empty as represented by the empty list (`[]`) specified for each node in the marking of RoutingTable.

The submodules of the DYMOProtocol module all need to access the routing table and the sequence number maintained by each node. To reduce the number of arcs in the modules, the routing table and the sequence numbers have been modelled using *fusion sets*. A fusion set allows a set of places in different modules to be linked together into one compound place across the hierarchical structure of the model. In this case, we have a fusion set OwnSeqNum (for linking together places modelling the sequence number of each node) and a fusion set RoutingTable (for linking the places modelling the routing table of each node). The name of the fusion set which a place belongs to (if any) is written in a tag positioned next to the place.

```
colset SeqNum          = int with 0 .. 65535;
colset NodexSeqNum     = product Node * SeqNum;
colset NodexSeqNumList = list NodexSeqNum;

colset RERRMessage = record HopLimit         : INT *
                            UnreachableNodes : NodexSeqNumList;

colset RoutingMessage = record TargetAddr : Node   * OrigAddr   : Nodes *
                               OrigSeqNum : SeqNum * HopLimit   : INT   *
                               Dist       : INT;

colset DYMOMessage = union RREQ : RoutingMessage + RREP : RoutingMessage+
                           RERR : RERRMessage;
```

Fig. 7. Colour set declarations for DYMO messages

```
colset RouteTableEntry = record
                           Address        : IPAddr * SeqNum : SeqNum *
                           NextHopAddress : IPAddr * Broken : BOOL   *
                           Dist           : INT;

colset RouteTable      = list RouteTableEntry;
colset NodexRouteTable = product Node * RouteTable;
```

Fig. 8. Colour set declarations for routing table entries

Initiate Route Discovery Module. We consider the InitiateRouteDiscovery module shown in Fig. 9 as a representative example of a submodule at the most detailed level of the CPN model. This module specifies how the route discovery procedure is initiated when a request for a route discovery arrives via the DYMORequest input port. The rectangles in Fig. 9 are ordinary transitions (i.e., non substitution transitions) which means that they can become *enabled* and *occur*. In the marking shown in Fig. 9, a token corresponding to a request for a route discovery originating at node 1 and targeting node 6 is present on the DYMORequest place. In this marking, the transition ProcessRouteRequest is enabled in the following *binding*:

⟨`rreq={originator=1,target=1}`⟩

which binds the *variable* `rreq` of colour set `RouteRequest` to the value in the `ROUTERREQUEST`. Evaluating the input arc expression on the arc from DYMORequest to ProcessRouteRequest results in a multi-set consisting of the single token present on place DYMORequest. The effect of an occurrence of ProcessRequest

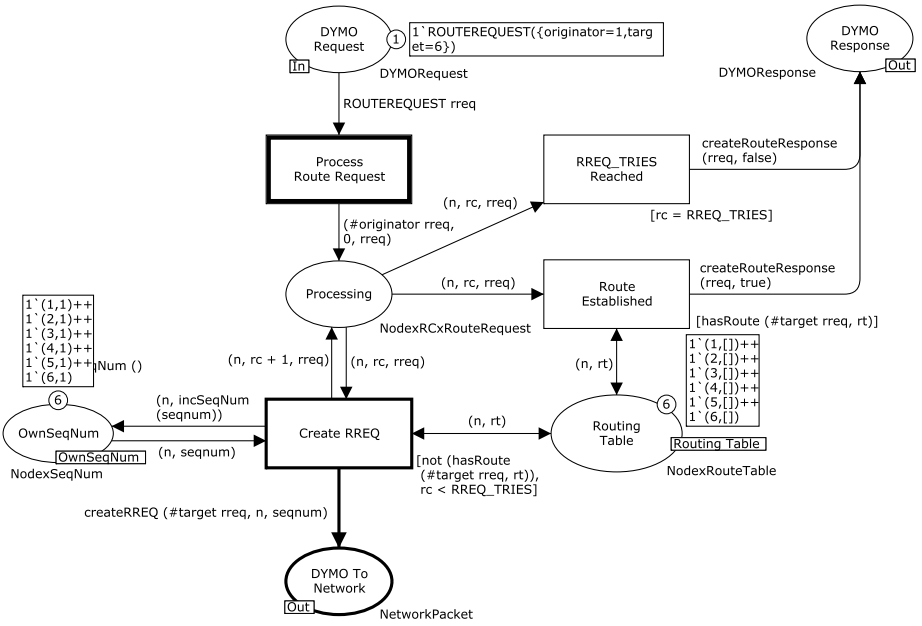


Fig. 9. The Initiate Route Discovery module - initial marking

with the binding above in the marking in Fig. 9 is that the token on DYMORequest is removed and a token is added to place Processing. The colour of the token added to Processing is obtained by evaluating the *arc expression* on the arc from ProcessRouteRequest to Processing in the binding from above:

```
(#originator rreq,0,rreq)
```

The SML operator `#originator` extracts the originator field in the value bound to `rreq`. The marking resulting from the occurrence of ProcessRouteRequest is shown in Fig. 10. A route request being processed is represented by a token on Processing over the colour set `NodexRCxRouteRequest` which is a product type. The first component of the token on Processing specifies the node processing the route request (i.e., the originator), the second component specifies how many times the RREQ has been retransmitted, and the third component specifies the route request.

In the marking shown shown in Fig. 10, the transition CreateRREQ is enabled with the binding:

$$\langle \texttt{rc=[],rreq=\{originator=1,target=1\},rc=0,n=1,seqnum=1} \rangle$$

The expression in square brackets positioned next to the CreateRREQ transition is a *guard* specifying an additional boolean conditions (beyond the presence of required tokens on input places) for the CreateRREQ transition to be enabled. In this case, the guard specifies that for the transition to be enabled, a route

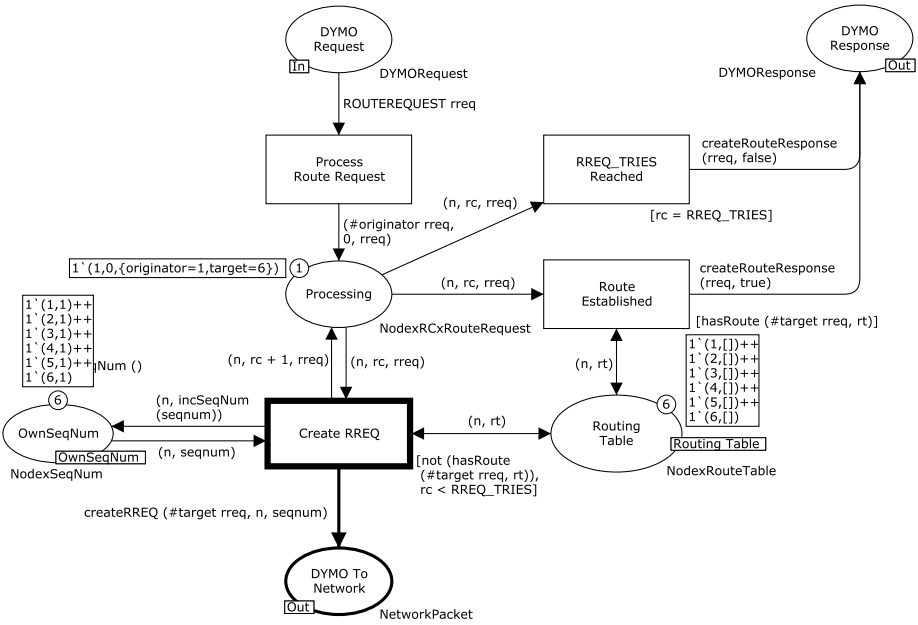


Fig. 10. The Initiate Route Discovery module - after ProcessRouteRequest occurrence

must not already exist in the route table `rt` to the target node `#target`, and the number of times `rc` the current route request has been retransmitted must be less than the retransmission limit `RREQ_TRIES` for RREQs. The SML function `hasRoute` used in the guard is implemented as follows:

```
fun hasRoute (target, rt:RouteTable) =
   List.exists (fn {Address, ...} => UNICAST(target) = Address) rt
```

and uses the predefined SML function `List.exists` to check whether an entry in the route table `rt` leading to the `target` node already exists. This is a typical example of how SML is used to represent (sequential) data manipulation.

The marking resulting from the occurrence of CreateRREQ is shown in Fig. 11. When sending a RREQ, the sequence number of node 1 sending the request is incremented by 1 and so is the counter specifying how many times the RREQ has been transmitted. Furthermore, a token corresponding to a network packet containing a RREQ message is produced on place DYMOToNetwork. The destination of the packet is set to `LL_MANET_ROUTERS` since it must be sent to all nodes within reach of node 1.

If a route becomes established (i.e., the originator receives a RREP for the RREQ), the RouteEstablished transition becomes enabled and a token can be put on place DYMOResponse indicating that the requested route has been successfully established. If the retransmission limit for RREQs is reached (before a RREP is received), the RREQ_TRIES Reached transition becomes enabled and a

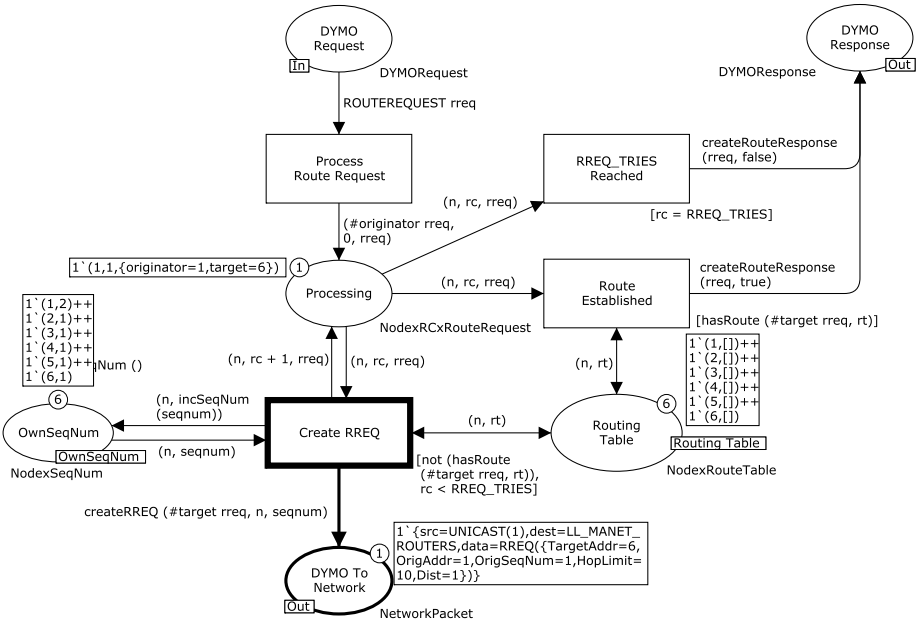


Fig. 11. The Initiate Route Discovery module - after CreateRREQ occurrence

token can be put on place DYMOResponse indicating that the requested route could not be established.

3.4 Modelling the DYMO Protocol Environment

The MobileWirelessNetwork module shown in Fig. 12 captures the mobile wireless network that DYMO is designed to operate over. It consists of two parts: one part modelling the transmission of network packets represented by the substitution transition WirelessPacketTransmission, and one part representing the mobility of the nodes represented by the Mobility substitution transition. The places DYMOToNetwork, NetworkToDYMO, and LinkState are associated to the similarly named socket places in Fig. 4. The transmission of network packets is done relative to the current topology of the MANET which is explicitly represented via the current marking of the Topology place. The topology is represented using the colour set `Topology` defined in Fig. 13.

The idea is that each node has an adjacency list of nodes that it can reach in one hop, i.e., its neighbouring nodes. The marking of place Topology in Fig. 12 corresponds to the topology in Fig. 1(left). This adjacency list is then consulted when a network packet is being transmitted from a node to determine the set of nodes that can receive the network packet. In this way, the dynamic topology is modelled by the addition and removal of nodes from the adjacency lists. The

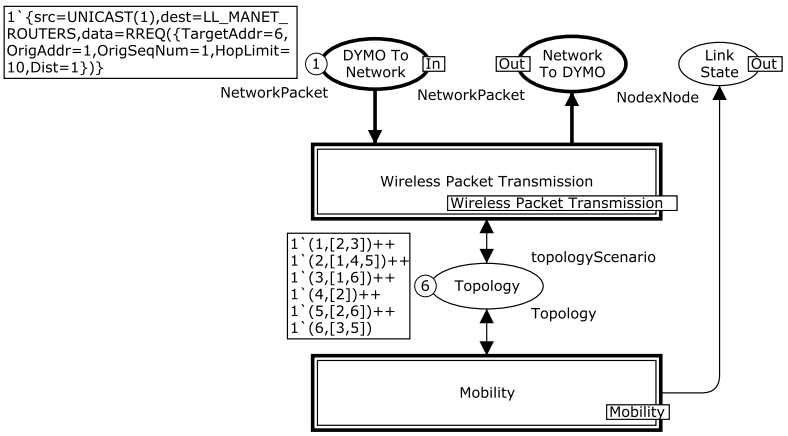


Fig. 12. The Mobile Wireless Network

```
colset NodeList = list Node;
colset Topology = product Node * NodeList;
```

Fig. 13. Colour set declarations for topology modelling

place LinkState models that a node can be informed about the reachability of its neighbouring nodes which is used in active link monitoring.

The WirelessPacketTransmission module models the actual transmission of packets and is shown in Fig. 14. The module captures how network packets are transmitted via the physical network from one node to the next. Packets are transmitted over the network according to the function transmit on the arc from the transition Transmit to the place NetworkToDYMO. When the Transmit transition occurs in a binding where the boolean variable `success` is set to `true`, then all nodes within reach of the sending node will receive the packet. Otherwise, no nodes will receive the packet. The transition Transmit is enabled in the marking shown in Fig. 14 (left) and the marking resulting from a successful transmission of the packet on DYMOToNetwork is shown in Fig. 14 (right). In this case two tokens are added to place NetworkToDYMO corresponding to nodes 2 and 3 receiving the packet being multi-casted from node 1.

In a real network, a transmission could be received by any subset of the neighbouring nodes (e.g., because of signal interference). Here it is only modelled that either all of the neighbouring nodes receive the packet or none of the nodes receive it. This is sufficient because the modelling of the dynamic topology means that a node can move out of reach of the transmitting node immediately

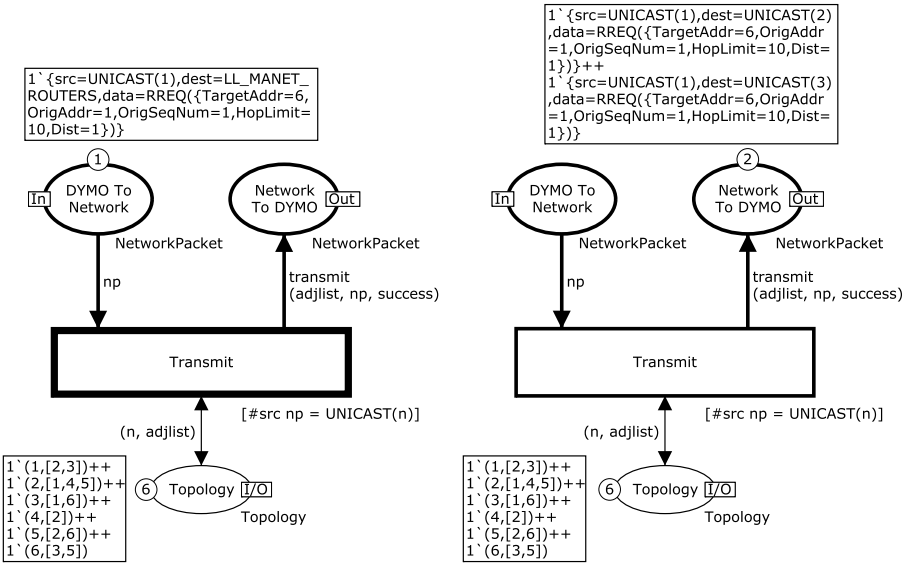


Fig. 14. Transmission of packets - before (left) and after (right) transmission

before the transmission occurs which has exactly the same effect as a signal interference in that the node does not receive the packet. Hence, signal interference and similar phenomena imply that a node does not receive a packet is in the model equivalent to the node moving out of reach of the transmitting node.

3.5 Lessons Learned and Perspectives

The development of the DYMO CPN model was based on the natural language specification provided in the Internet draft [15] specifying the DYMO protocol. The modelling work was done when version 10 [15] was the most recent DYMO specification. In the process of constructing the CPN model and simulating it, several issues and ambiguities in the specification were discovered. The most important ones are summarised in Table 1. These issues were submitted to the IETF MANET Working Group mailing list [82] and issue 1 and 3-7 were acknowledged by the DYMO developers and taken into account in the subsequent version DYMO specification [16] (version 11). Issue 2 was not considered critical as it causes route discovery to fail in scenarios which according to the experience of the DYMO developers would seldom occur in practise.

The modelling conducted with the DYMO protocol illustrates that the construction of a formal and executable model provides a very systematic and comprehensive way of reviewing a protocol design document (such as the DYMO

Table 1. DYMO CPN modelling [25]: issues identified in the modelling phase

Issue	Description
1	When processing a routing message, a DYMO router may respond with a RREQ flood, i.e., a RREQ addressed to the node itself, when it is target for a RREQ message (cf. [15], Sect. 5.3.4). It was not clear from the specification which information to put in the RREQ message, i.e., the originator address, hop limit, and sequence number of the RREQ.
2	When judging the usefulness of routing information, the target node is not considered. This means that a new request with a higher sequence number can make an older request for another node stale since the sequence number in the old message is smaller than the sequence number found in the routing table.
3	When creating a RREQ message the distance field in the message is set to zero. This means that for a given node n an entry in the routing table of a node n' connected directly to n may have a distance to n which is 0. Distance is a metric indicating the distance traversed before reaching n, and the distance between two directly connected nodes should be one.
4	In the description of the data structure route table entry (cf. [15], Sect. 4.1) it is suggested that the address field can contain more than one node. It was not clear why this was the case.
5	When processing RERR messages (cf. [15], Sect. 5.5.4) it is not specified whether the hop limit shall be decremented.
6	When retransmitting a RREQ message (cf. [15], Sect. 5.4), it was not explicitly stated whether the node sequence number should be increased.
7	Version 10 of DYMO introduced the concept of distance instead of hop count. Distance is a more general metric, but in the routing message processing (cf. [16], Sect. 5.3.4) it is incremented by one. We believe it should be up to the implementers how much distance is incremented depending on the metric used.

Internet draft) and how it can contribute to increasing the quality of a protocol design specification. Similar conclusions can also be drawn from other case studies where CPN modelling has been applied to protocols developed in the context of IETF. A CPN model of the DYMO protocol has also been developed in [12] where a considerably more compact CPN model of the DYMO protocol directly targeting state space exploration was developed. A number of other issues related to the functionality of the DYMO protocol were reported in [12]. In comparison to the CPN model in this section, the CPN model developed in [12] provides a more abstract modelling approach that does not use an explicit representation of MANET topology.

4 The GAN Protocol Architecture

This section focuses on how standard behavioural properties of CPNs in combination with explicit state space exploration can be used to verify basic properties of protocols. Furthermore, this section gives an example of how CPNs can be used to model a system spanning multiple protocols and protocol layers. The presentation is based on a project [30] in which CPN modelling and state space exploration was used at TietoEnator Denmark in early phases of developing an implementation corresponding to a particular instantiation [42] of the generic GAN architecture [2] aimed at integrating IP and telephone services.

4.1 GAN Secure Connection Establishment

The Generic Access Network (GAN) [2] architecture specified by the 3rd Generation Partnership Project (3GPP) [3] allows access to common telephone services such as SMS and voice-calls via IP networks. A central part of the GAN architecture is the establishment of a secure connection between a *mobile station* (e.g., a mobile phone) and a *GAN controller* through a *security gateway*. The GAN architecture relies on standardised protocols such as Dynamic Host Configuration Protocol (DHCP) for IP address configuration, IP Security (IPsec) [65] for encryption and authentication, and Internet Key Exchange v2 (IKEv2) protocol [64] for negotiation of IPsec parameters.

The purpose of the CPN model constructed in the project was two-fold. Firstly, to define the scope of the protocol software to be developed by TietoEnator. More specifically, the aim was to determine which parts of the generic GAN specification were to be included in the implementation to be developed by TietoEnator. Secondly, to specify the detailed design and usage of the involved protocol software components. The focus of the CPN model is on the establishment of a secure tunnel and the initial GAN message exchanges since this is where important details were not provided in the full GAN specification. In particular, the full GAN specification [2] contained no clear specification of the IKEv2 message exchange and the states that the protocol entities should be in when establishing a GAN connection (at the time of the project in 2007). Furthermore, the GAN specification only states that IKEv2 and IPSec are to be used, and in which operating modes.

4.2 CPN Model of the GAN Protocol Architecture

The CPN model of the secure connection establishment consists of 31 modules organised into four hierarchical levels. In the following, we present four selected modules from the CPN model. Our purpose is to illustrate how the phases that the protocol entities enter when establishing a GAN connection have been modelled, and provide sufficient detail on the CPN model in order for the reader to interpret the verification results presented later. A more in-depth presentation of the CPN model can be found in [30].

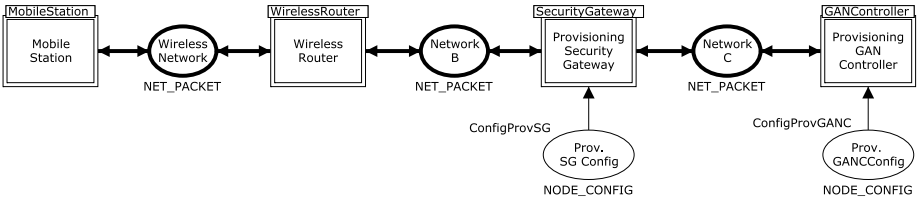


Fig. 15. Top-level module of the GAN model

Figure 15 shows the top-level module which is organised so that it mimics the GAN network architecture. The substitution transition MobileStation represents the mobile station which is connecting to the telephone network via an IP network. The place Wireless Network connected to MobileStation represents the wireless network which connects the mobile station to a wireless router represented by the substitution transition WirelessRouter. The wireless router is an arbitrary access point with routing functionality, and is connected to the Provisioning Security Gateway, through NetworkB. As part of establishing a GAN connection, an encrypted tunnel is established between the mobile station and the security gateway. The encrypted tunnel is provided by the Encapsulating Security Payload (ESP) mode of the IP security layer (IPSec) [65]. To provide such an encrypted tunnel, both ends have to authenticate each other, and agree on both an encryption algorithm and keys. This is achieved using the Internet Key Exchange v2 (IKEv2) protocol [64]. The provisioning security gateway is connected to the Provisioning GAN Controller via NetworkC. The GAN controllers are connected to the telephone network and perform the relay of traffic to/from the IP networks (NetworkC and the WirelessNetwork). This in turn allows mobile stations to access the services on the telephone network. The places with thin lines connected to the substitution transitions Provisioning Security Gateway and Provisioning GAN Controller are used to provide configuration information to the corresponding network nodes. The CPN model does not include modelling of the telephone network as the scope of the CPN model covers the components involved in establishing the connection with the GAN controller. Furthermore, as the purpose of the model was to represent the protocol entities present on each of the nodes in the network architecture, it sufficed that the model encompassed one mobile node, one wireless router, one provisioning security gateway, and one provisioning GAN controller.

The basic exchange of messages in establishing a GAN connection to the provisioning GAN controller involves three steps. The first step is for the mobile station to acquire an IP address on the wireless network using DHCP. The second phase is to create a secure tunnel to the provisioning security gateway. Having established the secure tunnel, the third phase is for the mobile station to open a secure connection to the GAN controller and register itself. Figure 16 (left) shows the IKEInitiator module of the mobile station and Fig. 16 (right) shows the IKEResponder module of the security gateway. These two peer modules

model the second step of the GAN connection establishment concerned with creating the secure tunnel. Incoming IP packets for the module arrive via the ReceiveBuffer input port places. Outgoing IP packets are put in the SendBuffer places. The states (phases) that the protocol entities goes through during the IKE message exchange when establishing the secure tunnel are represented by the places connecting the substitution transitions.

The state changes are represented by substitution transitions. The submodules of the substitution transitions specify the processing rules for messages during the individual phases. Figure 17 shows the Send IKE_SA_INIT Packet and Handle_SA_INIT_Request modules which are the submodules of the two top-most substitution transitions in Fig 16. The Send IKE_SA_INIT Packet transition in Fig. 17 (left) takes the IKE Initiator from the state Ready to Await IKE_SA_INIT and sends an IKE message to the security gateway initialising the communication. The IP address of the security gateway is retrieved from the Ready place. Figure 17 (right) shows how the IKE_SA_INIT packet is handled by the IKE Responder. The guard of the HandleSA_INIT_Request transition ensures that the transition is only enabled if the incoming packet (token) on IncomingIKERequest represents a IKE_SA_INIT packet. In that case, it sends an IKE packet back to the initiator as specified by the arc expression on the arc from Handle_SA_INITRequest and the responder enters the Wait for EAP Auth state. The submodules of the other substitution transitions in Fig. 16 are similar.

The establishment of a GAN connection involves multiple layers of the IP network stack. DHCP (used to configure the mobile station) and GAN are application layer protocols, IKE is a transport layer protocol, and IPSec belongs to the network layer. As a consequence, the CPN model of GAN connection spans multiple protocol layers. Furthermore, the protocol entities also access and manipulate the *routing table* and a *security policy database* (SPD) which is maintained at the IP network layer. The establishment of a GAN connection accesses the routing table of a node in order to ensures that packets are put into the secure tunnel, and extracted again at the other end. The SPD describes what packets are allowed to be sent and received by the IP protocol stack, and is also responsible for identifying which packets are to be tunnelled at the mobile station and the security gateway. Each entry in the SPD contains the source and destination addresses to use for matching packets, and an action to perform. Modelled actions are *bypass* (which means allow packet to pass without tunnelling) and *tunnel* (the matched packet is to be sent through an ESP tunnel). As we will see later, the content of the routing table and the SPD play an important role in validating the correctness of the GAN connection establishment. It was therefore required to explicitly represent them in the CPN model.

4.3 Verification of the GAN CPN Model

The goal of applying state space exploration was to verify the completeness of the design. This included verifying that all phases, steps, and messages involved in establishing a secure GAN connection were covered by the design, and the correctness of the connection establishment,i.e., that a GAN connection is

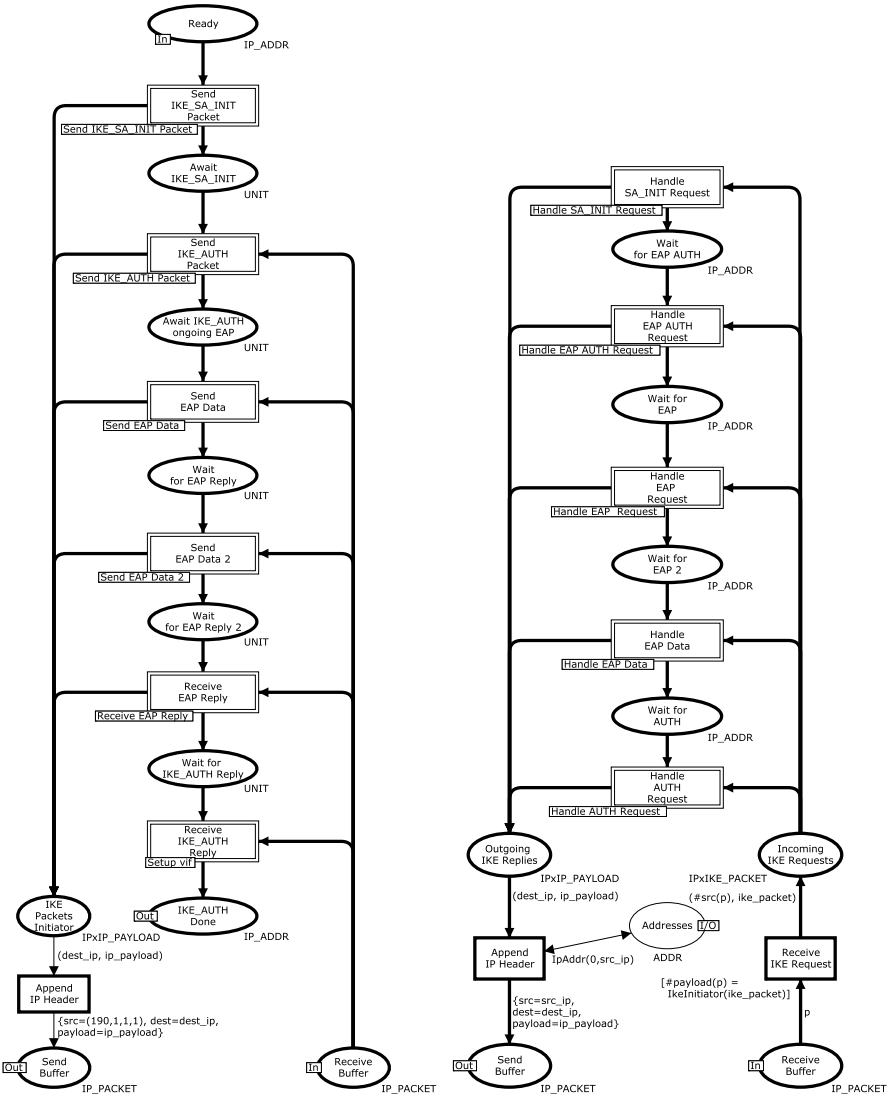


Fig. 16. IKE initiator (left) and IKE responder (right) modules

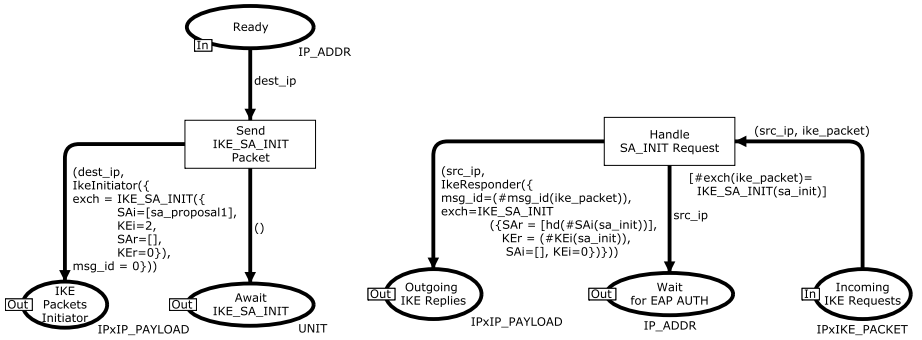


Fig. 17. Example of IKE initiator (left) and IKE responder (right) submodules

eventually established with the mobile station and the GAN controller being properly synchronised. Verification of the key properties of the design for secure connection establishment was done by *state space exploration*. The basic idea underlying state space exploration is to compute all reachable states and state changes of the CPN model and represent these as a directed graph, where nodes represent markings and arcs represent occurring binding elements. State spaces can be constructed fully automatically by the state space tool in CPN Tools. Verification of the GAN scenario modelling by means of state spaces relied on the use of the *state space report* that can be generated by CPN Tools. The generation of a state space report for the smallest possible configuration of a considered protocol is typically the first step performed when conducting verification of a CPN model.

The state space report is divided into several sections. In the following we present excepts from the individual sections and explain how they can be used for the verification. Figure 18 shows the first part of the state space report for the CPN model. This part provides some *state space statistics* specifying how large the state space is. It can be seen that the state space consists of $3,854$ nodes and $9,225$ arcs. The construction of the state space took 4 seconds. Statistics for the strongly connected component graph (SCC-graph) are also specified. It has $3,514$ nodes and $8,881$ arcs, and was calculated in 2 seconds. The fact that there are fewer nodes in the SCC-graph than in the state space implies that there are non-trivial strongly connected components (SCCs), i.e., SCCs consisting of more than a single state space node. This means that infinite executions exist and that the GAN connection establishment may not terminate. We will investigate the reasons for this at the end of this subsection.

The *boundedness properties* section of the state space report specifies how many and which tokens a place may hold – when considering all reachable states (markings). Figure 19 lists the best upper and best lower integer bounds for selected places in the mobile station module. It can be seen that the first four places modelling the states of the mobile station contain at most one token and may contain zero tokens. Similarly, it can be seen that there is at most one token

```
State Space                    Scc Graph
  Nodes:   3,854                 Nodes:   3,514
  Arcs:    9,225                 Arcs:    8,881
  Secs:        4                 Secs:        2
```

Fig. 18. State space report – statistics

```
Best Integer Bounds        Upper      Lower
   Down                     1          0
   Ready                    1          0
   VIF open to Prov. SG     1          0
   VIF Closed               1          0

   Send Buffer              1          0
   Receive Buffer           1          0
   Network Buffer           1          0

   Routing Table            1          0
   Security Policy Database 1          1
```

Fig. 19. State space report – integer bounds

in the send, received, and network buffers. The place RoutingTable has a lower integer bound of 0 and an upper integer bound of 1. The lower integer bound is 0 since in the initial marking there are no tokens on this place. During the start-up procedure of the mobile station, a token representing a list of routing table entries is put on this place. The place SecurityPolicyDatabase has a best upper and a best lower integer bound of 1. This means that there is always exactly one token present on this place. This is because the security policy database is modelled as a single token being a list containing the current entries in the security policy database.

The *best upper multi-set bound* of a place specifies for each colour in the colour set of the place the maximal number of tokens that is present on this place with the given colour in any reachable marking. This is specified as a multi-set, where the coefficient of each value is the maximal number of tokens with the given value. If the coefficient is zero, then the colour is omitted in the specification. Figure 20 shows part of the state space report providing the upper multi-set bounds for the security policy databases of the mobile station, wireless router, security gateway, and the GAN controller. The upper multi-set bounds specify the possible tokens that can reside on these places and by carefully inspecting these bounds it was possible to validate that the possible entries in the security

```
Mobile Station: Security Policy Database
    1‘[{src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=PayloadList([PAYLOAD_DHCP]),policy=SpdBypass}]++
    1‘[{src=((80,1,1,1),32),dest=((12,0,0,0),8),
        nl_info=AnyNextLayer,policy=ESPTunnel(((190,1,1,1),(172,1,1,2)))},
       {src=((190,1,1,1),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]++
    1‘[{src=((190,1,1,1),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]

Wireless Router: Security Policy_Database
    1‘[{src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]

Security Gateway: Security Policy Database
    1‘[{src=((13,0,0,0),8),dest=((80,1,1,1),32),
        nl_info=AnyNextLayer,policy=ESPTunnel(((172,1,1,2),(190,1,1,1)))},
       {src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]

GAN Controller: Security Policy Database
    1‘[{src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]
```

Fig. 20. State space report – best upper multi-set bounds

policy database were all as desired. Altogether, an inspection of the boundedness properties helped significantly in increasing confidence in the correctness of the design in terms of proper settings of the routing table and the security policy database.

Figure 21 shows the part of the state space report specifying the *home and liveness properties*. The home properties show that there exists a single *home marking*, which is state number 3854. A home marking is a state which can be reached from any reachable state. For the GAN scenario model this means that it is impossible to have an execution sequence starting from the initial state (initial marking) which cannot be extended to reach state 3854. The liveness properties tell us that there is a single *dead marking* which is also state number 3854. A dead marking is a state in which no transitions are enabled. This means that the marking corresponding to node 3854 is both a home and a dead marking.

To obtain information about the marking corresponding to node number 3854, the node number was transferred into the simulator of CPN Tools and displayed graphically on the CPN model. It was then checked (by inspecting the markings of the individual places) that the marking corresponded to the desired terminating state of the GAN connection establishment procedure, i.e., the state where

```
Home Properties                 Liveness Properties
  Home Markings: [3854]           Dead Markings: [3854]
```

Fig. 21. State space report – home and liveness properties

the mobile station has obtained an IP address, has successfully communicated with the provisioning GAN controller, all protocol modules are in a state corresponding to the GAN connection having been established, and the routing tables and security databases contain the correct entries. The fact that state 3854 is the only dead marking tells us that the protocol as specified by the CPN model is partially correct, i.e., if execution terminates we have the correct result. Furthermore, because node 3854 is also a home marking it is always possible to terminate the GAN connection establishment with the correct result.

The analysis above showed that it is always possible to terminate the GAN connection establishment procedure correctly, but there is no guarantee that it will eventually happen. The section of the state space report providing information about *fairness properties* showed that the two transitions RejectDiscoveryRequest and HandleGARCReject which are part of the GAN controller module were *impartial*. This means that these two transitions occur infinitely often in any infinite occurrence sequence. The two transition occurs if the GAN controller decides to reject an incoming connection from a mobile station. Hence, if the connection establishment procedure does not terminate in the single home and dead marking identified, then it is because the GAN controller keeps rejecting the connection.

4.4 Lessons Learned and Perspectives

The validation of secure connection establishment in the considered GAN scenario is representative for how validation of protocols is typically performed with CPN Tools – as it in practise involves a combination of both simulation and state space exploration. As part of the construction of the GAN model, the support for interactive simulation in CPN Tools was used to perform detailed checks to ensure that the model behaviour was as desired. Even though the use of interactive simulations (and simulation in general) cannot be used to prove correct behaviour, it proved to be very useful in identifying situations related to improper manipulations of the entries in the routing tables and security policy database - or when additional detail not present in the GAN specification had to be worked out and specified. Furthermore, interactive simulation was helpful in identifying issues that led the GAN connection establishment procedure to terminate prematurely, e.g., because a certain phase of the connection establishment was missing in the CPN model. These issues manifested themselves in markings where the GAN connection had not yet been established, but where

no transitions were enabled. This was in particular effective in making explicit where further specification of the message exchanges were required.

The interactive simulation was in later phases replaced with automatic simulation where a number of random executions of the CPN model were performed with the purpose of checking whether the execution of the CPN model resulted in a state in which the GAN connection was properly established. Eventually state space exploration of the CPN model was conducted which succeeded in establishing the key property that a GAN connection will eventually be established provided that the GAN controller does not keep rejecting the connection request. The verification conducted also illustrated the general observation that in many cases, the use of basic state space exploration and the state space report (i.e., investigating standard behavioural properties of Petri nets) are sufficient in establishing key properties of a protocol design. In this case the state space was small in size and could be generated in a few seconds without the use of advanced state space exploration techniques.

5 The Routing Interoperability Protocol

The section show how a CPN model can be augmented with application-specific behavioural visualisation reflecting the execution of the CPN model. This section is based on a project conducted at Ericsson Telebit A/S addressing the specification of the Routing Interoperability Protocol (RIP)[1] for routing packets between IP core networks and mobile ad-hoc networks. The CPN model of RIP augmented with behavioural visualisation was used as an early model-based prototype of RIP. It allowed the protocol design to be discussed among protocol engineers unfamiliar with CPNs, and it also enabled the protocol design to be presented to customers with the purpose of soliciting requirements of the services to be provided by the protocol.

5.1 CPN Model of the RIP Protocol

The main purpose of the routing interoperability protocol is to ensure that a packet flow between a host in the core network and a mobile node in an ad-hoc network is always relayed via one of the closest gateways that connect the core network and the mobile ad-hoc network. Figure 22 shows the top level module of the CPN model which reflects the network architecture that the RIP protocol is designed to operate in. The network architecture consists of three parts: an IPv6 core network represented by the CoreNetwork substitution transition (left) and its submodules, a mobile ad-hoc network represented by the AdHocNetwork substitution transition (right) and its submodules, and two gateways represented by the substitution transitions Gateway1 and Gateway2. The basic idea in the interoperability protocol is that the mobile nodes register the IPv6 address in the Domain Name Server (DNS) server in the core network that corresponds to

[1] RIP as discussed in this section should not be confused with the Routing Information Protocol[85].

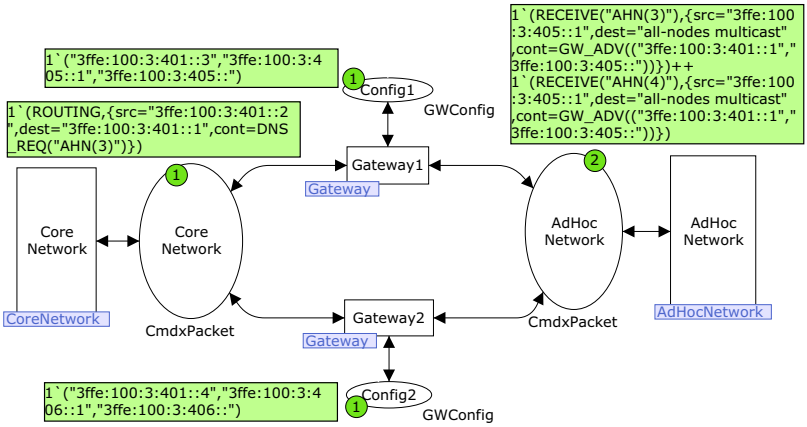


Fig. 22. The System module – top-level module of the CPN model

an IPv6 address prefix announced by the closest (preferred) gateway. Updates to the DNS database managed by the DNS server rely on the Dynamic Domain Name System Protocol [108].

The places CoreNetwork and AdHocNetwork are used for modelling the packets in transit on the core network and ad-hoc network, respectively. Figure 22 depicts a state in which there is one token on place CoreNetwork and two tokens on place AdHocNetwork. As an example, place CoreNetwork contains one token with the colour:

```
(RECEIVE("3ffe:100:3:401::1"), {src="3ffe:100:3:401::2",
       dest="3ffe:100:3:401::1",cont=DNSREQ("AHN(3)")})
```

representing a DNS request (`DNSREQ`) in transit on the core network from a host with source IPv6 address `3ffe:100:3:401::2` to a DNS server with destination IPv6 address `3ffe:100:3:401::1`. IPv6 addresses are 128-bit and by convention written in hexadecimal notation in groups of 16-bits separated by a colon (:). Leading zeros are skipped within each group and a double colon (::) is a shorthand for a sequence of zeros. Addresses consist of an *address prefix* and an *interface identifier*.

The place AdHocNetwork contains two tokens representing gateway advertisements in transit to nodes in the ad-hoc network. The gateways periodically announce their presence to nodes in the mobile ad-hoc network by sending *gateway advertisements* containing an IPv6 *address prefix*. The two Config places contain a token representing the configuration of the corresponding gateway. It consists of the IPv6 address of the gateway interface connected to the core network, the IPv6 address of the gateway interface connected to the ad-hoc network, and the address prefix announced by the gateway. Address prefixes are written in the form x/y where x is an IPv6 address and y is the length of the prefix. The mobile nodes in the ad-hoc network configure IPv6 addresses based

on the received gateway advertisements. In the marking depicted in Fig. 22, Gateway1 is announcing the 64-bit address prefix 3ffe:100:3:405::/64 and Gateway2 is announcing the prefix 3ffe:100:4:406::/64. Each of the gateways has configured an address on the interface to the ad-hoc network based on the prefix they are announcing to the ad-hoc network. Gateway1 has configured the address 3ffe:100:3:405::1 and Gateway2 has configured the address 3ffe:100:3:406::1. The gateways have also configured addresses on the interface to the core network based on the 3ffe:100:3:401::/64 prefix of the core network.

Figure 23 lists the definitions of the colour sets used in the System module. IP addresses, prefixes, and symbolic IP addresses are represented by colour sets IPAdr, Prefix, and Symname all defined as the set of strings. The colour set PacketCont and Packet are used for modelling the IP packets. The five different kinds of packets used in RIP are modelled by the PacketCont colour set:

DNS_REQ modelling a DNS request packet. It contains the symbolic IP address to be resolved to a (numerical) IP address by a DNS server.

DNS_REP modelling a DNS reply. It contains the symbolic IP address and the resolved IP address.

DNS_UPD modelling a DNS update. It contains the symbolic IP address to be updated and the new IP address to be bound to the symbolic address.

GW_ADV modelling the advertisements disseminated from the gateways. An advertisement contains the IP address of the gateway and the announced prefix.

```
colset Prefix = string;  (* address prefixes *)
colset IPAdr = string;   (* IP addresses     *)
colset SymName = string; (* symbolic names   *)

colset SymNamexIPAdr = product SymName * IPAdr;
colset IPAdrxPrefix = product IPAdr * Prefix;

colset PacketCont = union DNS_REQ : SymName +       (* DNS Request     *)
                          DNS_REP : SymNamexIPAdr + (* DNS Reply       *)
                          DNS_UPD : SymNamexIPAdr + (* DNS Update      *)
                          GW_ADV  : IPAdrxPrefix +  (* Advertisments   *)
                          PACKET;                   (* Generic payload *)

colset Packet = record src  : IPAdr  * dest : IPAdr * cont : PacketCont;
colset Cmd    = union ROUTING              + RECEIVE      : IPAdr +
                      FLOODING     : IPAdr + GWAHNROUTING : IPAdr +
                      AHNGWROUTING : IPAdr;

colset CmdxPacket = product Cmd * Packet;
colset GWConfig   = product IPAdr * IPAdr * Prefix;
```

Fig. 23. Colour set definitions used in the System module

`PACKET` modelling generic payload packets belonging to packet flows between hosts and the mobile nodes.

The colour set `Packet` models the packets as a record containing the source, destination, and content. The actual payload (content) and layout of packets are not essential for modelling the interoperability protocol and has therefore been abstracted away. The colour set Cmd is used to control the operation of the various modules in the CPN model. The colour set GWConfig models the configuration information of the gateway.

The Core Network. Figure 24 shows the CoreNetwork module modelling the core network. This module is the immediate submodule of the substitution transition CoreNetwork of the System module shown in Fig. 22. The port place CoreNetwork is assigned to the CoreNetwork socket place in the System module (see Fig. 22). The substitution transition Routing represents the routing mechanism in the core network. The substitution transition Host represents the host on the core network, and the substitution transition DNS Server represents the DNS server that maintains the DNS database.

The Mobile Ad-hoc Network. Figure 25 depicts the AdHocNetwork module modelling the mobile ad-hoc network. The place Nodes is used to represent the nodes in the mobile ad-hoc network. The place RoutingInformation is used to represent the routing information in the ad-hoc network which is assumed to be available via some routing protocol executed in the ad-hoc network. This routing information enables among other things the nodes to determine the distance to the reachable gateways. Detailed information about the colour of the token on place RoutingInformation has been omitted.

Figure 26 lists the definition of the colour sets used in the AdHocNetwork module. The colour set AHNConfig is used to model the configuration information for the mobile ad-hoc nodes. Each ad-hoc node is represented by a token on place Nodes and the colour of the tokens specifies the name of the node and a list of configured IP addresses. Each configuration specifies the IP address configured, and the IP address and prefix of the corresponding gateway. It is possible for

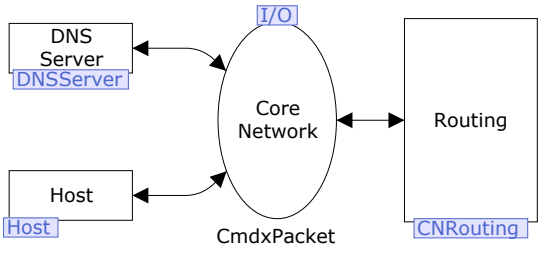


Fig. 24. Core Network module – modelling the core network

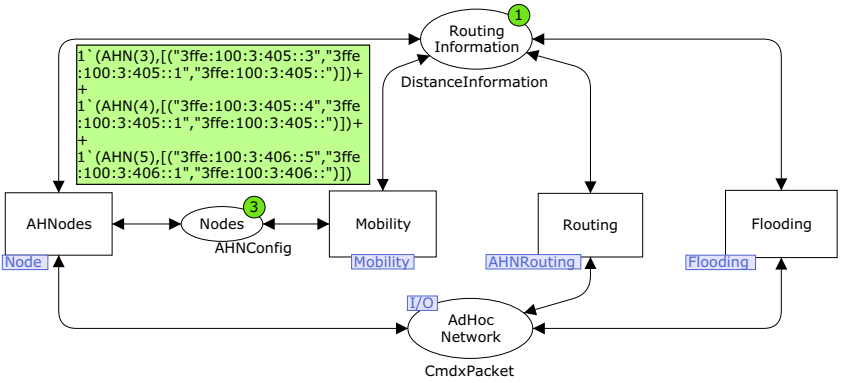


Fig. 25. AdHocNetwork module – modelling the ad-hoc network

```
(* --- ad-hoc nodes --- *)
color AHId = int with 1..5;
color AHNode = union AHN : AHId;

(* --- configuration information for ad-hoc nodes --- *)
color AHNIPConfig = product IPAdr * IPAdr * Prefix;
color AHNIPConfigs = list AHNIPConfig;
color AHNConfig = product AHNode * AHNIPConfigs;
```

Fig. 26. Colour definitions used in the AdHocNetwork module

a mobile ad-hoc node to configure an IP address for multiple gateways. The mobile node must ensure that the DNS database always contains the IP address corresponding to the *preferred gateway*. In the marking shown in Fig. 25, it can be seen from the labels below the mobile nodes that Ad-hoc Node 3 and Ad-hoc Node 4 have configured IP addresses based on the prefix announced by Gateway1, whereas Ad-Hoc Node 5 has configured an IP address based on the prefix announced by Gateway2. For an example, Ad-hoc Node 3 has configured the address 3ffe:100:3:405::3.

There are four substitution transitions in the AdHocNetwork module corresponding to the components of the ad-hoc network. The substitution transition AHNodes represents the behaviour of the nodes in the mobile ad-hoc network. The substitution transition Mobility models the mobility of nodes in the ad-hoc network, i.e., that the nodes may move closer or further away from the gateways. The substitution transition Routing represents the routing protocol executed in the ad-hoc network. The purpose of the routing protocol in the context of the RIP protocol is to provide the nodes with information about distances to the

gateways. The routing is abstractly modelled in a similar way as the routing mechanism in the core network and will not be discussed further in this paper. The substitution transition Flooding models the dissemination of advertisements from the gateways. A detailed presentation of this part of the model has been omitted here. The complete CPN model of the RIP protocol is hierarchically structured into 18 modules. A detailed presentation of the CPN model can be found in [74].

5.2 Behavioural Visualisation of the RIP Protocol

In the routing interoperability project, the BRITNeY Suite animation framework [111] was used to create an *animation GUI* on top of the CPN model. The animation GUI allows a user to observe the execution of the constructed CPN model using a graphical representation of the network architecture. The graphics is updated by the underlying CPN model according to the execution of the formally specified protocol, and the CPN model is also able to react to stimuli provided by the user via the animation GUI.

Figure 27 shows a representative snapshot of the application-specific graphics during the execution of the CPN model. The IP addresses configured by the individual nodes are shown as labels below the nodes. For an example, Ad-hoc Node 3 has configured two IP addresses: 3ffe:100:3:405:3 and 3ffe:100:3:406:3. The convention is that the preferred IP address is the topmost address in the list below the node. The entries in the DNS database are shown in the upper left corner. It shows the entries for each of the three ad-hoc nodes. The two numbers written at the top of each node are counters that provide information about the number of packets on the incoming (left) and outgoing (right) interfaces of the nodes. Transmissions of advertisements from the gateways are visualised by green dots. Fig. 27 shows an example where Gateway2 is transmitting an advertisement. Transmission of payload packets is visualised using red dots, and DNS packets are visualised using blue dots.

In addition to observing feedback on the execution of the CPN model in the animation GUI, it is also possible to provide input to the CPN model directly via the animation GUI. The user can move the nodes in the ad-hoc network thereby changing the distances to the two gateways. It is also possible to define a packet flow from the host in the core network to one of the nodes in the mobile ad-hoc network by clicking on the red square positioned next to each of the ad-hoc nodes. The square will change its colour to green once the CPN model has registered the flow. The flow can be stopped again by clicking on the (now green) square next to the mobile ad-hoc node. Finally, it is possible to force the transmission of an advertisement from a gateway by clicking on the gateway.

A more generic form of high-level graphical feedback in the form of MSCs was also used in this project. Figure 28 shows an example of an MSC diagram based on a simulation of the CPN model. The MSC shows a scenario where Ad-hoc Node 3 makes a Move and discovers that Gateway 2 is now the closest gateway. This causes it to send a DNS update to the DNS server. The last part of the MSC shows the host initiating a packet flow to Ad-hoc Node 3. One benefit of

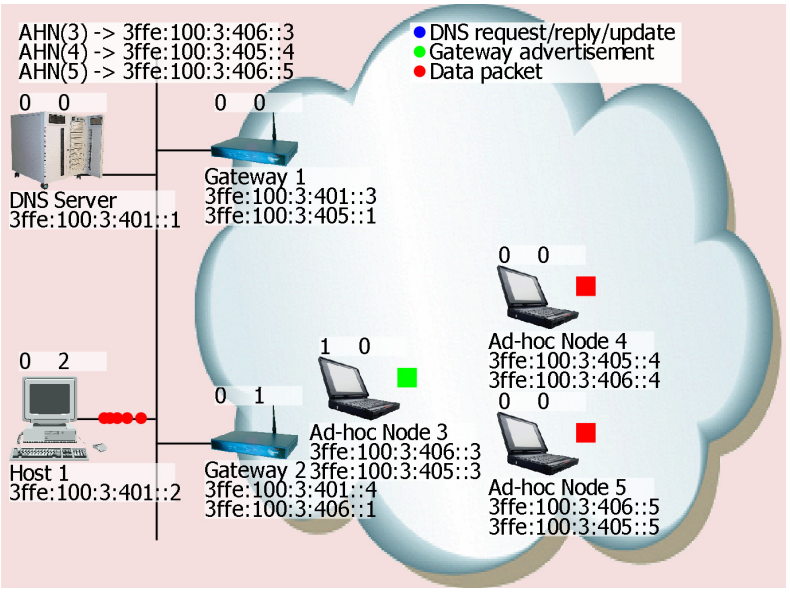


Fig. 27. Snapshot of the interaction graphics

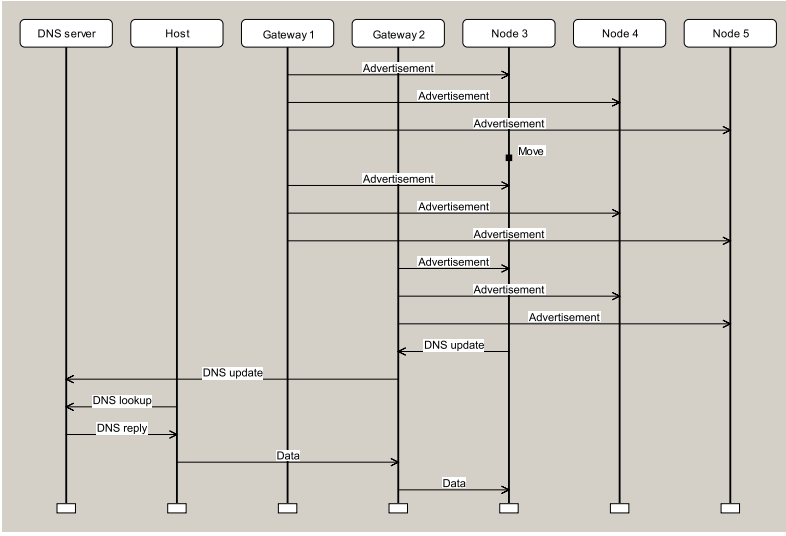


Fig. 28. Message sequence chart generated by the animation GUI

using MSCs is that they provide an event-based view that records the execution history. This is in contrast to the state-based view on the CPN model that one obtains during an interactive simulation. The two forms of feedback therefore complement each other and MSCs have been widely used in projects where CPNs were applied to protocol design.

Graphical feedback from the execution of the CPN model is achieved by attaching *code segments* to the transitions in the CPN model. These code segments are sequential pieces of code that are executed whenever the corresponding transition occurs in the simulation/execution of the CPN model. As an example consider the CNRouting module in Fig. 29. The transition Route models the routing of the packet on the core network. It uses the routing information on place RoutingInformation to direct the packet to the proper gateway. The SML function FindNextHop in the guard expression of the transition computes the IP address of the next hop gateway using the routing information and destination IP address of the packet. The Route transition has an attached code segment which is executed whenever the transition occurs. The code segment invokes the primitives in the animation package for animating the transmission of packets in the core network.

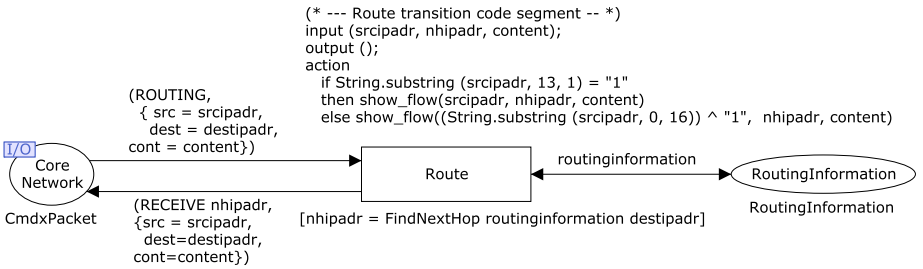


Fig. 29. The CNRouting module – Routing in the core network

The CPN model receives input from the animation GUI by polling the animation GUI for events. An event queue has been implemented between the animation GUI and the CPN model. The code segment of transition Produce in the Poll module shown in Fig. 30 polls the animation GUI for events at regular intervals during the execution of the CPN model. Events are put into a list-token representing an event queue on the place Events. The parts of the CPN model that are to react on events from the animation GUI are linked via place fusion to the Event place and are able to consume events from the event queue. The occurrence of the transition Produce corresponds to a poll to the animation GUI for events.

5.3 Lessons Learned and Perspectives

The CPN model combined with the animation GUI that was developed in the RIP project served as an early model-based executable prototype. The domain

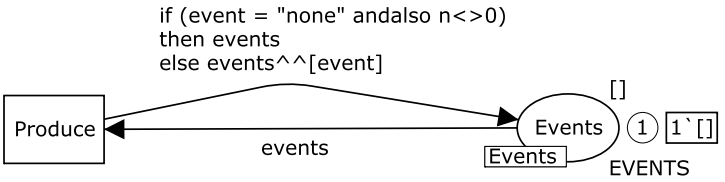


Fig. 30. The Poll module – Polling the animation GUI for events

specific graphical user interface (the animation GUI) made it possible to explore and demonstrate the design of the interoperability protocol with the underlying formal model being transparent for the observer and the demonstrator. In particular, it made it possible for persons without knowledge of the CPN modelling language to experiment with the proposed design. The use of an animation GUI on top of the CPN model has the advantage that the behaviour observed by the user is as defined by the underlying model that formally specifies the design. The alternative would have been to implement a separate visualisation application totally detached from the CPN model. This would have led to double representation of the dynamics of the interoperability protocol which could in turn lead to inconsistencies between the two representation of the design.

Another advantage offered by the development of a model-based prototype is ease of control compared to a physical prototype, in particular in the case of mobile nodes and wireless communication where scenarios can be very difficult to control and reproduce. The use of a model means that there is no need to invest in physical equipment and there is no need to setup the actual physical equipment early in the project. The use of a model also makes it possible to investigate larger scenarios, e.g., scenarios that may not be feasible to investigate with the available physical equipment. An additional general advantage of the approach taken in the RIP project is that at an early stage of development, the implementation details can be abstracted away and only the key part of the design have to be specified in detail. As an example, the CPN model of the interoperability protocol abstracted away the routing mechanisms in the core and ad-hoc networks, and the mechanism used for distribution of advertisements. Instead, the service assumed from these components for the interoperability protocol to work was modelled. The possibility of making abstraction means that it is possible to obtain an executable prototype without implementing all the components.

6 The Edge Router Discovery Protocol

The previous sections have demonstrated how modelling, simulation, state space exploration, and behavioural visualisation can be applied for validating the functional design of protocols. This section summarises a project [67] conducted with Ericsson Telebit A/S where a combination of the techniques introduced in the previous sections were applied for the design of an Edge Router Discovery Protocol (ERDP). The CPN model of ERDP was developed in close cooperation

with the protocol engineers at Ericsson Telebit A/S based on a natural-language specification that would normally have served as a basis for the implementation of the protocol. Simulation and MSCs were used in initial investigations of the ERDP protocol behaviour. Then state space exploration was used to conduct a formal verification of the key behavioural properties of ERDP. The aim of this section is to show how modelling, simulation, visualisation, and state space exploration all can help to identify omissions and behavioural errors in a design, and how they are typically used in conjunction in a protocol design process.

6.1 CPN Model of the ERDP Protocol

ERDP is based on the IPv6 Neighbour Discovery Protocol (NDP) [88] and supports *edge routers* residing on the boundary of an *IP core network* in configuring *gateways* with an IPv6 address prefix. This address prefix can in turn be used by mobile nodes in ad-hoc networks to configure global IPv6 unicast addresses and obtain Internet access via the core network. Figure 31 shows the ERDP module which is the top-level module of the CPN model. The substitution transition Gateway represents the gateway, and the substitution transition EdgeRouter represents the edge router. The wireless communication link between the edge router and the gateway is represented by the substitution transition GW_ER_Link. The four socket places GWIn, GWOut, ERIn, and EROut model packet buffers between the link layer and the gateway and edge router. Both the gateway (GW) and the edge router (ER) have an incoming and an outgoing packet buffer.

All four places in Fig. 31 have the colour set IPv6Packet, used to model the IPv6 packets exchanged between the edge routers and gateways. Since ERDP is based on the IPv6 Neighbour Discovery Protocol, the packets are carried as Internet Control Message Protocol (ICMP) packets. The definitions of the colour sets for NDP, ICMP, and IPv6 packets were derived directly from RFC 2460 [22] by using record type constructors for representing fields within packets

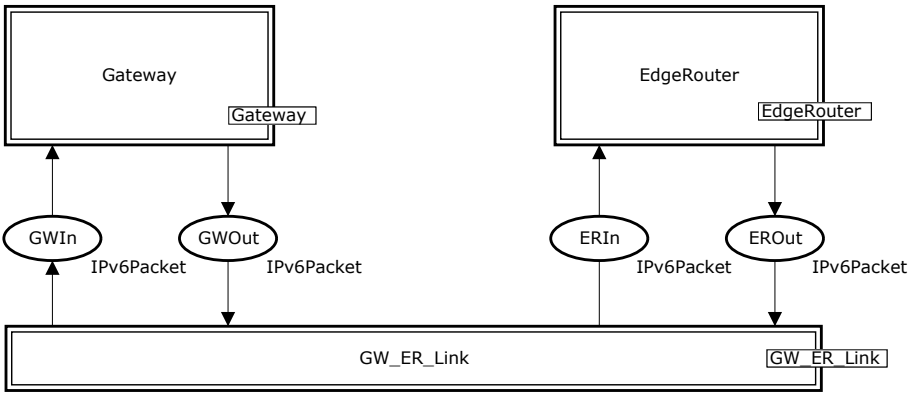


Fig. 31. Top-level module of the ERDP CPN model

and union type constructors for representing the different kinds of packets (see [67] for detail). It was considered important by the protocol engineers for later implementation that the definition of the packets followed closely the structure of IPv6 packets instead of a more abstract representation.

Figure 32 shows the EdgeRouter module. The port places ERIn and EROut are related to the accordingly named socket places in the ERDP module (see Fig. 31). The place Config models the configuration information associated with the edge router, and the place PrefixCount models the number of prefixes still available in the edge router for distribution to gateways. The place PrefixAssigned is used to keep track of which prefixes are assigned to which gateways.

Figure 33 shows the declarations of the colour sets for the three places in Fig. 32. The configuration information for the edge router (modelled by the colour set ERConfig) is a record consisting of the IPv6 link-local address and the link-layer address of the edge router. A list of pairs (colour set ERPrefixAssigned) consisting of a link-local address and a prefix is used to keep track of which prefixes are assigned to which gateways. A counter modelled by the place PrefixCount with the colour set PrefixCount is used to keep track of the number of prefixes still available. When this counter reaches 0, the edge router has no further prefixes available for distribution. The number of available prefixes can be modified by changing the initial marking of the place PrefixCount, which is set to 1 by default.

The substitution transition SendUnsolicitedRA (in Fig. 32) corresponds to the multicasting of periodic *unsolicited router advertisements* (RAs) by the edge router such that gateways can discover the presence of the edge router. When a gateway receives an unsolicited RA, it responds with a unicast *router solicitation* (RS). The substitution transition ProcessRS models the reception at the edge

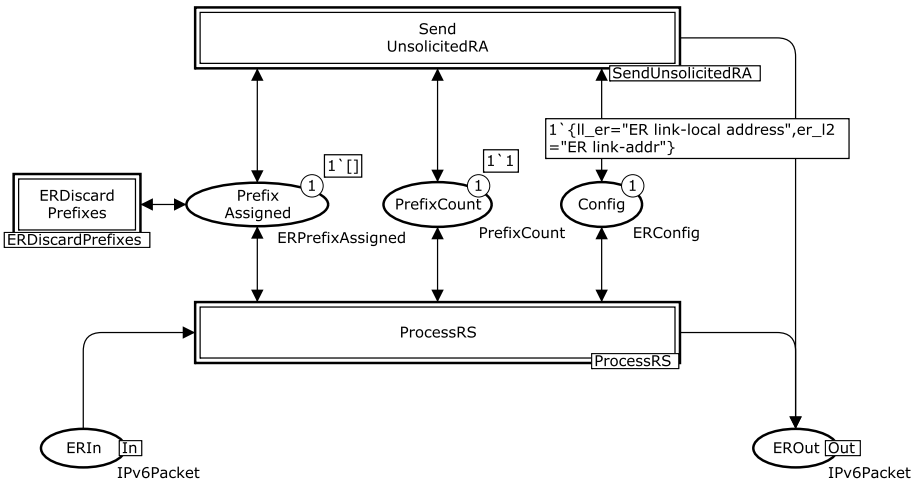


Fig. 32. The EdgeRouter module

```
colset LinkAddr    = string;

colset ERConfig = record ll_er : IPv6Addr * (* link-local address  *)
                         er_l2 : LinkAddr;  (* link-addr (layer 2) *)

colset ERPrefixEntry    = product IPv6Addr * IPv6Prefix;
colset ERPrefixAssigned = list ERPrefixEntry;

colset PrefixCount = int;
```

Fig. 33. Colour set definitions for edge routers

router of unicasted RSs from gateways, and the sending of a unicast RA to the gateway in response. The substitution transition ERDiscardPrefixes models the expiration of prefixes on the edge router side.

The marking shown in Fig. 32 has a single token on each of the three places used to model the internal state of the edge router protocol entity. In the marking shown, the token on the place PrefixAssigned with the colour [] corresponds to the edge router not having assigned any prefixes to the gateways. The token on the place PrefixCount with colour 1 indicates that the edge router has a single prefix available for distribution. Finally, the colour of the token on the place Config specifies the link-local and link addresses of the edge router. In this case the edge router has the symbolic link-local address ER link-local address, and the symbolic link-address ER link-addr.

Figure 34 depicts the SendUnsolicitedRA module which is the submodule of the substitution transition SendUnsolicitedRA in Fig. 32. The transition SendUnsolicitedRA models the sending of the periodic unsolicited router advertisements. The variable erconfig is of type ERConfig, and the variable prefixleft is

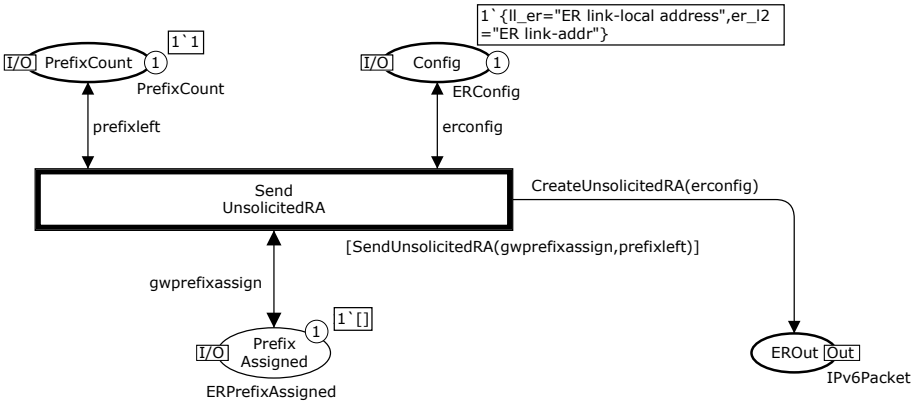


Fig. 34. Initial marking of the SendUnsolicitedRA module

of type PrefixCount. The transition SendUnsolicitedRA is enabled only if the edge router has prefixes available for distribution, i.e., prefixleft is greater than 0. This is ensured by the function SendUnsolicitedRA in the guard of the transition.

Figure 35 depicts the marking of the SendUnsolicitedRA module after the occurrence of the transition SendUnsolicitedRA in the marking shown in Fig. 34. An unsolicited router advertisement has been put in the outgoing buffer of the edge router. It can be seen that the `DestinationAddress` is the address `all-nodes-multicast`, the `SourceAddress` is `ER link-local address`, and the `LinkLayerAddress` (in the options part) is `ER link-addr`.

Figure 36 shows the part of the GW_ER_Link module that models transmission of packets from the edge router to the gateway across the wireless link. Transmission of packets from the gateway to the edge router is modelled similarly. The port places GWIn and EROut are linked to the similarly named socket places in Fig. 31. The transition ERtoGW models the successful transmission of packets, whereas the transition LossERtoGW models the loss of packets. The variable ipv6packet is of type IPv6Packet. A successful transmission of a packet from the edge router to the gateway corresponds to moving the token modelling the packet from the place EROut to GWIn. If the packet is lost, the token will only be removed from the place EROut.

Wireless links, in general, have a lower bandwidth and higher error rate than wired links. These characteristics have been abstracted away in the CPN model since the purpose is to reason not about the performance of ERDP but rather its logical correctness. Duplication and reordering of messages are not possible on typical one-hop wireless links, since the detection of duplicates and the preservation of order are handled by the data-link layer. The modelling of the wireless links does allow overtaking of packets, but this overtaking was elimi-

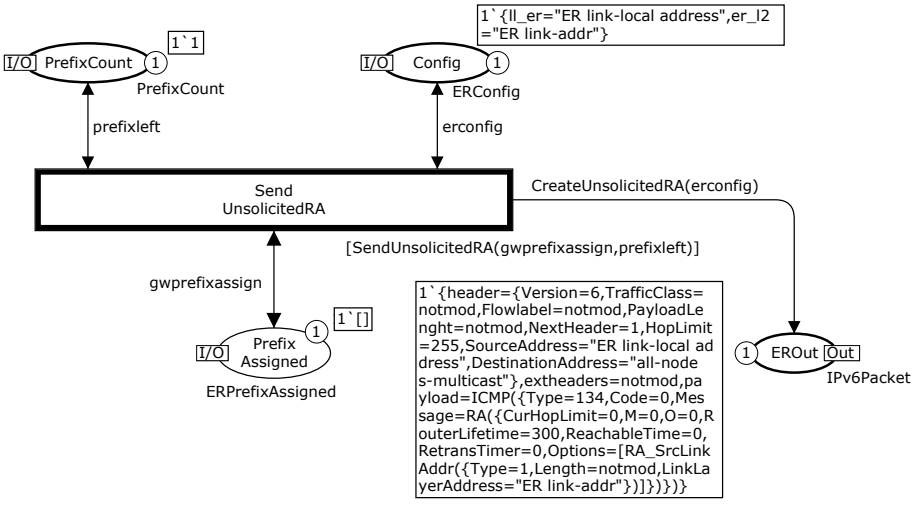


Fig. 35. Module SendUnsolicitedRA, after occurrence of SendUnsolicitedRA

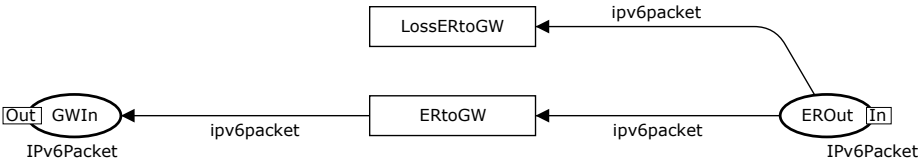


Fig. 36. Part of the GW_ER_Link module

Table 2. ERDP project [67] – design issues identified in the modelling phase

Category	Review 1	Review 2	Total
Errors in protocol specification/operation	2	7	9 issues
Incompleteness and ambiguity in specification	3	6	9 issues
Simplifications of protocol operation	2	0	2 issues
Additions to the protocol operation	4	0	4 issues
Total	11	13	24 issues

nated in the state space exploration phase where bounds were imposed on the capacity of the input and output packet buffers.

The CPN model was developed as an integrated part of the development of ERDP. The creation of the CPN model was done in cooperation with the protocol engineers at Ericsson in parallel with the development of the ERDP specification. Altogether 70 person-hours were spent on CPN modelling. Prior to the development of the CPN model, the protocol engineers at Ericsson were given a 6 hour course on CPNs that made them capable of reading CPN models. This means that CPN models could be used actively in discussion related to the design of the ERDP protocol. MSCs (to be illustrated shortly), integrated with simulation were used in both review steps to investigate the behaviour of ERDP in detail. The use of MSCs in the project was of particular relevance since it presented the operation of the protocol in a form well known to the protocol engineers. Altogether 24 design issues were identified during three iterations on the CPN model. Table 2 categorises and enumerates the issues encountered during two review phases (Review 1 and Review 2) of the protocol design. The issues were identified in the process of constructing the CPN model, performing single-step executions of the CPN model, and engaging in discussions of the CPN model with the protocol engineers at Ericsson.

6.2 Verification of the ERDP CPN Model

State space exploration was conducted after the three iterations of modelling as discussed in the previous section. The purpose of the state space exploration was to conduct a more thorough investigation of the operation of ERDP, including verification of its key properties. The key behavioural property of ERDP is

proper configuration of the gateway with prefixes. This means that for a given prefix and state where the gateway has not yet been configured with that prefix, the protocol must be able to configure the gateway with that prefix. Furthermore, when the gateway has been configured with the prefix, the edge router and the gateway should be *consistently configured*, i.e., the assignment of the prefix must be recorded both in the gateway and in the edge router protocol entity. Whether a marking represents a consistently configured state for a given prefix can be checked by inspecting the marking of the place PrefixAssigned in the edge router and the marking of the place Prefixes in the gateway.

Obtaining a finite state space. The first step towards state space exploration of the CPN model was to obtain a finite state space. The CPN model as presented above has an infinite state space, since an arbitrary number of tokens (packets) can be put on the places modelling the packet buffers. As an example, the edge router may initially send an arbitrary number of unsolicited router advertisements. To obtain a finite state space, an upper integer bound of 1 was imposed on each of the places GWIn, GWOut, ERIn, and EROut (see Fig. 31) which model the packet buffers. This also prevents overtaking among the packets transmitted across the wireless link. Furthermore, the number of packets simultaneously present in the four input/output buffers was limited to 2. Technically, this was done by using the *branching options* available in the CPN state space tool to prevent the processing of enabled transitions whose occurrence in a given marking would violate the imposed bounds on the buffer places.

No packet loss and prefix expire. The second step was to consider the simplest possible configurations of ERDP, starting with a single prefix and assuming that there is no packet loss on the wireless link and that prefixes do not expire. The full state space for this configuration had 46 nodes and 65 arcs. Inspection of the state space report showed that there was a single dead marking represented by node 36. Inspection of this node showed that it represented a state where all of the packet buffers were empty. However, the edge router and gateway were inconsistently configured in this state in that the edge router had assigned the prefix P1 (the single prefix), while the gateway was not configured with that prefix. This was an error in the protocol. To locate the source of the problem, query functions in the state space tool were used to obtain a counter example leading from the node representing the initial marking to node 36. Figure 37 shows the resulting error trace, visualised by means of an MSC. This MSC was generated automatically from the extracted counter example. The column labelled GW-Buffer represents the packet buffer between the gateway protocol entity and the underlying protocol layers. Similarly, the ERBuffer column represents the packet buffer in the edge router. The problem is that the edge router sends two unsolicited RAs. The first one gets through and the gateway is configured with the prefix, which can be seen from the event marked with *A* in the lower part of the MSC. However, when the second RS, without any prefixes, is received by the edge router (the event marked with *B*), the corresponding solicited RA will not contain any prefixes. Because of the way the protocol was specified, the

gateway will therefore update its list of prefixes to the empty list (the event marked with *C*), and the gateway is no longer configured with a prefix.

To fix the error, the protocol was modified so that the edge router always replies with the list of all prefixes that it has currently assigned to the gateway. The state space for the modified protocol consisted of 34 nodes and 49 arcs, and there were no dead markings in the state space. The state space report specified that there were 11 home markings. Inspection of these 11 markings showed that they all represented consistently configured states for the prefix P1. The markings were contained in the single terminal SCC of the state space. A terminal SCC is an SCC of the state space where all successors of states in the SCC belong to the SCC itself. This shows that, from the initial marking it is always possible to reach a consistently configured state for the prefix, and that when such a marking has been reached, the protocol entities will remain in a consistently configured state. To verify that a consistently configured state would eventually be reached, it was checked that the single terminal SCC was the only non-trivial SCC. A trivial SCC is a SCC consisting of just a single state. This showed that all cycles in the state space (which correspond to non-terminating executions of the protocol) were contained in the single terminal SCC, which (from above) contained only consistently configured states. The reason why the protocol is not supposed to terminate in a consistently configured state represented by a dead marking is that the gateway may, at any time, when it is configured, send a router solicitation back to the edge router to have its prefixes refreshed.

Increasing the number of prefixes. When the correctness of the protocol had been established for a single prefix, the number of prefixes was increased. When there is more than one prefix available it no longer holds that a marking will *eventually* be reached where *all* prefixes are consistently configured. The reason is that with more than one prefix, the edge router may at any time decide not to configure the gateway with additional prefixes. Hence, a state where all prefixes have been consistently configured might not eventually be reached. Instead, firstly, it was verified that there was a single terminal SCC, all markings of which represent states where all prefixes have been consistently configured. This shows that it is always possible to reach such a marking, and when the protocol has consistently configured all prefixes, the protocol entities will remain consistently configured. Secondly, it was checked that all markings in each non-trivial SCC represented markings where the protocol entities were consistently configured with a subset of the prefixes available in the edge router. The properties above was checked using a number of user-defined queries in the state space tool of CPN Tools.

Adding packet loss. The third step was to allow packet loss on the wireless link between the edge router and the gateway. First, the case was considered in which there is only a single prefix for distribution. The state space for this configuration had 40 nodes and 81 arcs. Inspection of the state space report showed that there was a single dead marking. This marking represented an undesired terminal state where the prefix had been assigned by the edge router, but the gateway

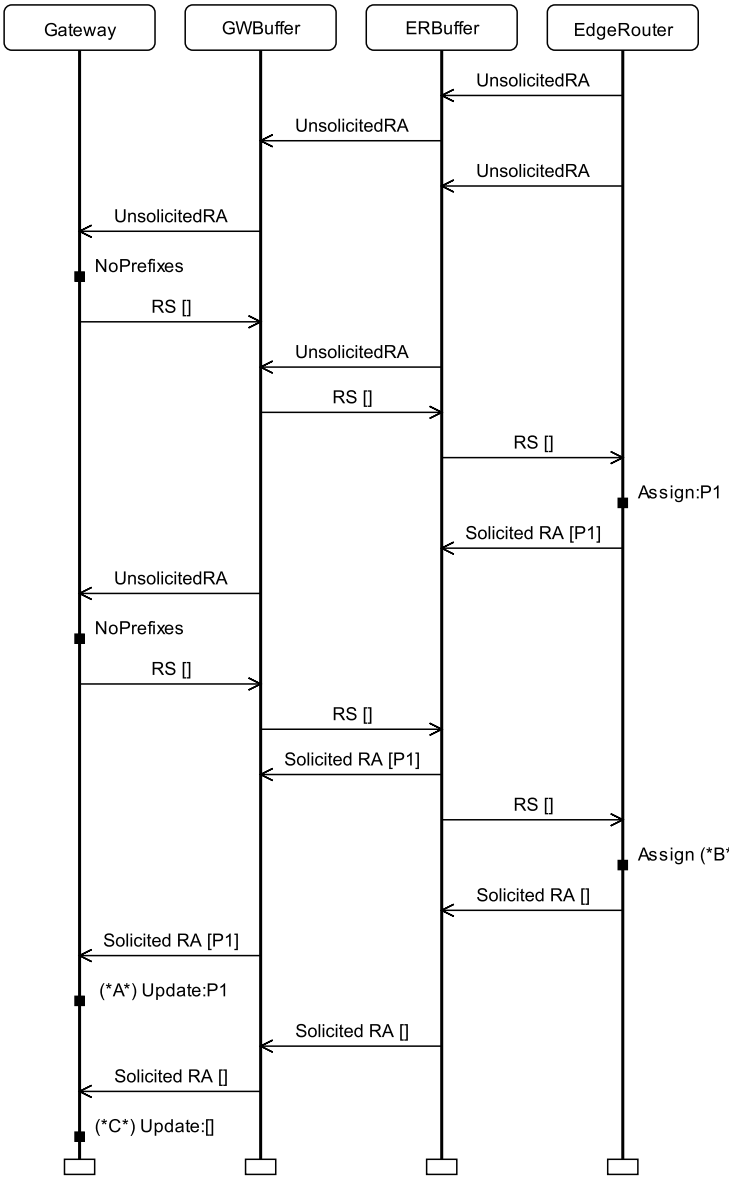


Fig. 37. MSC showing an execution leading to an undesired terminal state

was not configured with the prefix. The source of the problem was located by extracting a counter example and visualising it in a similar manner as shown in Fig. 37. The problem was fixed by ensuring that the edge router would resend an unsolicited RA to the gateway as long as it had prefixes assigned to the gateway. The state space of the revised CPN model had 68 nodes and 160 arcs. Inspection of the state space report showed that there were no dead markings and no home markings. Investigation of the terminal SCCs showed that there were two terminal SCCs, each containing 20 markings. The nodes in one of them all represented states where the edge router and gateway were consistently configured with the single prefix P1, whereas the nodes in the other terminal SCC all represented states where the protocol entities were not consistently configured. The markings in the undesired terminal SCC represent a livelock in the protocol, i.e., if one of the markings in the undesired terminal SCC is reached, it is no longer possible to reach a state where the protocol entities are consistently configured with the prefix. The source of the livelock was related to the control fields used in the router advertisements for refreshing prefixes and their interpretation on the gateway. This was identified by obtaining the MSC for a path leading from the initial marking to one of the markings in the undesired terminal SCC. As a result, the processing of router advertisements in the gateway was modified. The state space for the protocol with the modified processing of router advertisements also had 68 nodes and 160 arcs. The state space had a single terminal SCC containing 20 nodes, which all represented states where the protocol entities were consistently configured with the single prefix.

When packet loss is present, it is not immediately possible to verify that the two protocol entities will eventually be consistently configured. The reason is that any number of packets can be lost on the wireless link. Each of the non-trivial SCCs was inspected using a user-defined query to investigate the circumstances under which the protocol entities would not eventually be consistently configured. This query checked that either all nodes in the non-trivial SCC represented consistently configured states or none of the nodes in the SCC represented a consistently configured state. For those non-trivial SCCs where no node represented a consistently configured state, it was checked that all cycles contained the occurrence of a transition corresponding to loss of a packet. Since this was the case, it can be concluded that any failure to reach a consistently configured states will be due to packet loss only. Hence, if finitely many packets are lost, a consistently configured state for some prefix will *eventually* be reached.

Adding prefix expire. The fourth and final step in the analysis was to allow prefixes to expire. The analysis was conducted first for a configuration where the edge router had only a single prefix to distribute. The state space for this configuration had 173 nodes and 513 arcs. The state space had a single dead marking, and inspection of this dead marking showed that it represented a state where the edge router has no further prefixes to distribute, it has no prefixes recorded for the gateway, and the gateway is not configured with any prefix.

This marking is a desired terminating state of the protocol, as we expect prefixes to eventually expire. Since the edge router has only finitely many prefixes to distribute, the protocol should eventually terminate in such a state. The single dead marking was also a home marking, meaning that the protocol can always enter the expected terminal state. When prefixes can expire, it is possible that the two protocol entities may never enter a consistently configured state. The reason is that a prefix may expire in the edge router (although this is unlikely) before the gateway has been successfully configured with that prefix. Hence, we are only able to prove that for any marking where a prefix is still available in the edge router, it is possible to reach a marking where the gateway and the edge router are consistently configured with that prefix.

Table 3 lists statistics for the size of the state space in the three verification steps for different numbers of prefixes. The column '|P|' specifies the number of prefixes. The columns 'Nodes' and 'Arcs' give the numbers of nodes and arcs, respectively, in the state space. For the state spaces obtained in the first verification step, it can be seen that 38 markings and 72 arcs are added for each additional prefix. The reason for this is that ERDP proceeds in phases where the edge router assigns prefixes to the gateway one at a time. Configuring the gateway with an additional prefix follows exactly the same procedure as that for the assignment of the first prefix. Once the state space had been generated, the verification of properties could be done in a few seconds. It is also worth observing that as the assumptions are relaxed, i.e., we move from one verification step to the next, the sizes of the state spaces grow. This, combined with the identification of errors in the protocol even in the simplest configuration, without packet loss and without expiration of prefixes, shows the benefit of starting state space exploration from the simplest configuration and gradually lifting assumptions. Furthermore, the state explosion problem was not encountered during the verification of ERDP, and the key properties of ERDP were verified for the number of prefixes that were envisioned to appear in practise.

6.3 Lessons Learned and Perspectives

The project at Ericsson highlights the benefits of a formal modelling and validation approach. Furthermore, the project emphasised the benefits of the model construction phase which is often underestimated (or not reported) in literature on protocol validation. As illustrated by the ERDP project, the modelling phase itself lead to significant insight into the protocol design, and contributed to a simpler and more complete protocol design. The construction of a CPN model and subsequent state space exploration can be seen as a very thorough and systematic way of reviewing the ERDP design specification. The project showed that the process of constructing a CPN model based on the ERDP specification provided valuable input to the ERDP design, and the use of simulation added further insight into the operation of the protocol. State space exploration, starting with the simplest possible configuration of the protocol, identified additional errors in the protocol. The results from state space exploration also

Table 3. State space statistics for the three verification steps

	No loss/No expire		Loss/No Expire		Loss/Expire	
|P|	Nodes	Arcs	Nodes	Arcs	Nodes	Arcs
1	34	49	68	160	173	531
2	72	121	172	425	714	2 404
3	110	193	337	851	2 147	7 562
4	148	265	582	1 489	5 390	19 516
5	186	337	926	2 390	11 907	43 976
6	224	409	1 388	3 605	23 905	89 654
7	262	481	1 987	5 185	44 550	169 169
8	300	553	2 742	7 181	78 211	300 072
9	338	625	3 672	9 644	130 732	505 992
10	376	697	4 796	12 625	209 732	817 903

demonstrate that errors are often present in the smallest configurations of a protocol system.

Using an iterative process where both a conventional natural-language specification and a CPN model were developed (as in this project) turned out to be an effective way of integrating CPN modelling and validation into the development of a protocol. In general, the combination of an executable formal model (such as a CPN model) and a natural-language specification seems to be provide a useful way to develop a protocol. One reason why both are required is that the software engineers that are eventually going to implement the protocol (which may be different from those that design the protocol) in many cases will not be familiar with the CPN modelling language. Secondly, in many cases there are important implementation elements of the protocol specification that are not reflected in the CPN model, such as the layout of packets.

It can be argued whether or not the issues and errors discovered in the process of modelling and conducting state space exploration would have been identified if additional conventional reviews of the ERDP specification had been conducted. Some of them probably would have been, but more subtle problems such as the inconsistent configurations discovered during state space exploration would probably not have been discovered until the first implementation of ERDP was operational. The reason for this is that discovering these problems requires one to consider subtle execution sequences of the protocol.

Overall, the application of CPNs in the development of ERDP was considered a success for three main reasons. Firstly, it was demonstrated that the CPN modelling language and supporting computer tools were powerful enough to specify and verify a real-world protocol being developed in an industrial project, and that integration into the conventional protocol development process is not difficult. Secondly, the act of constructing the CPN model, executing it, and discussing it led to the identification of several non-trivial design errors and

issues that, under normal circumstances, would not have been discovered until, at best, the implementation phase. Finally, the effort of constructing the CPN model and conducting state space exploration was represented by approximately 100 person-hours. This is a relatively small investment compared with the many issues that were identified and resolved early as a consequence of constructing and analysing the CPN model.

7 Related Work on CPN Protocol Validation

The four protocol examples presented in this paper constitute only a small subset of the examples that have been published in the literature on the use of CPNs for specification and validation of protocols - in particular in relation to protocols developed in the context of IETF and other protocol standardisation bodies.

The Datagram Congestion Control Protocol (DCCP) developed by the IETF has been investigated in [11]. DCCP is intended to provide an unreliable transport service with congestion control mechanisms. The work in [11] was done in parallel with the development of the emerging DCCP standard, and concentrated on modelling and verification of the connection establishment and synchronisation procedures of DCCP. It resulted in the identification of several functional errors in the protocol design, including discovery of deadlocks, non-progress behaviour (chatter), and problems with connection establishment in relation to sequence number wraps. The formal validation resulted in the IETF working group making small (but important) changes to the connection establishment and synchronisation procedures of DCCP. The work also included the development of a formal service specification for DCCP [33] and application of the sweep-line method [105] for on-the-fly checking of the protocol conformance to the developed service specification.

The classical Transmission Control Protocol (TCP) has also been modelled and verified using CPNs [10]. Similar to the work on DCCP, this work concentrated on the connection establishment procedures. It resulted in verifying the absence of deadlocks and livelocks in connection establishment, and a detailed specification of the circumstances under which TCP connection establishment may not be successful. Another example of transport layer protocol modelling and validation can be found in [104] which considers the Stream Transmission Control Protocol (SCTP).

The Internet Open Trading Protocol (IOTP) designed to provide an interoperability framework for Internet commerce was formally modelled and validated using CPNs in [90–92]. IOTP is designed to handle common trading procedures and encompass trading roles such as consumer, merchant, payment handler, and delivery handler. IOTP is organised around a collection of eight baseline transactions consisting of Purchase, Refund, Value exchange, Authentication, Withdrawal, Deposit, Inquiry, and Ping. These transactions comprise a minimal set of transactions for an Internet commerce protocol. A formal specification of the service provided by IOTP was developed using CPN in [92]. The service

was specified in the form of a finite-state automaton labelled with service primitives. The automaton was extracted from the state space of the CPN model by identifying the binding elements corresponding to service primitives of the protocol. A CPN model of the IOTP protocol itself was presented in [90, 91]. State space exploration focused on the termination properties and absence of livelocks in the IOTP transactions. The use of state space exploration revealed deficiencies related to termination of transactions. A verification of the IOTP protocol CPN model [90, 91] against the formal service specification from [92] was presented in [89]. Finite-state automata language comparison was used as the criterion for conformance following the methodology of [9]. Application of the sweep-line state space method on IOTP was investigated in [34] exploiting an inherent progression from the start of an IOTP transaction to termination of the transaction.

The Wireless Application Protocol (WAP) has been considered in [40, 41]. WAP is designed to provide Internet services to a wide range of hand-held devices. The work of [40, 41] concentrates on the Wireless Transaction Protocol (WTP) which is an important element of the WAP architecture and protocol suite. The work in [40] presents a formal modelling of the WTP service and a formal modelling of the WTP protocol. Checking the conformance of the WTP protocol against the WTP service was done using finite-state automata language comparison. This approach succeeded in detecting several inconsistencies between the protocol and the service which was provided as input to the WAP forum responsible for the development of WAP. The sweep-line method was used in [41] to alleviate the state explosion problem and allow for the verification of larger configurations of WTP. The application of the sweep-line method allowed configurations with parameter settings of retransmission counters corresponding to the recommended setting for GSM and IP network to be verified.

The Session Initiation Protocol (SIP) is a widely used protocol for the establishment of Internet multimedia session, and has been subject to formal modelling and validation in [23, 77]. The INVITE transactions have been formally analysed using state space exploration in [23, 77] leading to identification of undesired terminating states of the protocol when operating over an unreliable communication medium. Security aspects of SIP have been investigated in [78]. The work of [37] focuses on the formal modelling of a SIP-based protocol for multi-channel service oriented architectures. A formalisation of SIP with the purpose of providing a framework model for present architectures in mobile computing is presented in [36]. Another multimedia control protocol, the Capability Exchange Signalling (CES) protocol, has been formally modelled using CPNs and verified using state space exploration in [79]. The work on the CES protocol led to the identification of protocol errors in presence of sequence number wrap. Suggested changes were incorporated in a revised CPN model, and it was formally verified showing that the discovered errors have been eliminated.

The NEO protocol which is part of the distributed transactional object database management system NEOPPOD was investigated using high-level Petri

Nets in [17]. The Coloane environment was used for the construction of the models, and verification was performed using the CPN-AMI and Helena tools. The NEO protocol is used to coordinate data storage and retrieval in a decentralised and distributed system where data can be stored on a number of data nodes and data is accessed through the primary master node. The focus of [17] was on the protocol used for the election of the primary master node. The model of the election part of the NEO protocol consisted of eighteen modules. Since there existed no specification document for the protocol, the Petri net model was reverse-engineered from a prototype implementation. The validation process which relied on the use of state spaces discovered two flaws in the implementation of the protocol. These were subsequently provided to the software engineers responsible for the implementation of the protocol component.

The Resource Reservation Protocol (RSVP) was formally modelled and verified in [106, 107]. The modelling and verification concentrates on verifying the absence of deadlocks and livelocks in relation to the setup, maintenance and path release procedures of RSVP. In addition, a number of RSVP specific behavioural properties were investigated which considered in detail the internal state of the sender, router, and receiver protocol entities of the protocol. The main contribution of the work was the development of a formal specification of the RSVP path procedures. Another example on the modelling of routing protocols can be found in [76] which uses Mobile Petri Nets to construct a formal model of the Mobile IP protocol. Mobile IP allows transport layer connections to be preserved when mobile nodes change their point of attachment to the Internet. CPNs have also recently been used for the verification of security protocols. Privacy enhancing protocols were considered in [99], and [39] addresses the modelling and validation of PANA Authentication and Authorisation Protocol. Examples of protocols for which parametric verification has been pursued in the context of CPNs can be found in [31, 32].

8 Conclusions and Outlook

Functional validation of protocol designs is one of the main application areas of CPNs and supporting computer tools [28]. In this paper, we have surveyed a selection of recent projects on modelling and functional validation of industry relevant protocols. The examples demonstrate how the elements of protocols can be modelled using CPNs, and they illustrate how a combination of simulation, application-specific behavioural visualisation, and state space exploration is typically applied in protocol validation with CPNs. From a modelling perspective, the protocol examples have ranged from models representing two (or few) peer protocol entities (e.g., GAN, EDRP, and RIP) having an explicit representation in the net structure, to parameterised models capable of modelling an arbitrary number of peer protocol entities by setting a model parameter (e.g., DYMO). The latter was based on constructing a folded model where the identity of the protocol entities is encoded explicitly as part of the token colours. The CPN

models have also illustrated modelling at different protocols layer ranging from models operating at a single protocol layer (e.g., DYMO and ERDP) to protocol system design involving multiple protocol layers and protocols (e.g., GAN and RIP). An important aspect of the examples is that the process of modelling and conducting single step simulation is an important (but often underestimated) activity in the validation of a protocol design.

The main technique available for functional verification of CPN models is that of explicit state space exploration. The examples presented in this paper show how basic state space exploration combined with the generation of a state space report relying on a number of standard behavioural properties of Petri Nets, provides a light-weight approach which in many cases is an important step in verifying key properties of a protocol design. The main reason for the wide spread application of state space exploration has been the presence of mature computer tool support combined with the main advantages of state space exploration in terms of being a highly systematic approach, being able to provide counter examples, and allowing for a high degree of automation. The compact modelling of protocols enabled by CPNs has, in many cases, had the effect that the full state space can be explored for at least the smallest configuration of the considered protocol. The GAN and ERDP examples presented in this paper are concrete examples illustrating this. Practise have shown that the primary capability offered by the advanced state space methods is the possibility of verifying larger configurations of the protocol - and in some cases [71] the configurations of the system that are expected to occur in practise. The ERDP example considered in this paper is another example of this. Hence, despite the fact that explicit state space exploration methods requires one to conduct verification relative to a particular configuration of the protocol, the current suite of availably state space methods combined with the power of modern computing platforms in many situations allows for the practical validation of industrial-sized protocols.

While CPNs have been successfully applied to modelling and validating protocol designs, there has been relatively few attempts at using the constructed CPN models in an automated or semi-automated manner as a basis for the actual implementation of protocols. Some simulation-based approaches were used in [87] and [70] for generating server-side implementations. Here, the simulation code for the CPN model generated by CPN Tools was extracted, and after undergoing automatic modifications (e.g., linking the code to external libraries), the generated simulation code is used as the system implementation. A limitation of this approach is that the execution speed is affected because each step in the execution of the program involves the computation and execution of enabled transitions (as done by a CPN simulator) in order to determine the next state. Secondly, the approach ties the target platform to that of the CPN Tools simulator which may make the approach impractical for many application domains due to resource consumption of the CPN simulator. The SML/NJ compiler used for the simulator in CPN Tools has a large memory footprint making it ill-suited, e.g., for the domain of embedded systems. Some initial work on a

translation-based approach can be found in [73]. Here a restricted form of CPNs was used for obtaining an Erlang implementation of the DYMO routing protocol. The approach in [73] relies on the use of Process-Partitioned CPNs which enforces a detailed modelling of the protocol design which is very close to an implementation level model. An area that will be important as part of efforts in developing capabilities for automated code generation is the development of CPN protocol modelling methodology on which only limited research has been undertaken [18].

Acknowledgements. The authors acknowledge the contribution of Kristian L. Espensen and Mads. K. Kjeldsen in the project on the DYMO protocol, Paul Fleischer for his contribution in the GAN project, Michael Westergaard and Peder Christian Nørgaard for their contribution to the project on the RIP protocol, and Kurt Jensen for his contributions in the ERDP project.

References

1. ISO/IEC 15437. Information technology. Enhancements to LOTOS (E-LOTOS) (September 2001)
2. 3GPP. Digital Cellular Telecommunications System (Phase 2+); Generic Access to the A/Gb Interface; Stage 2. 3GPP TS 43.318 version 6.9.0 Release 6 (March 2007)
3. 3GPP. Website of 3GPP (May 2007), `http://www.3gpp.org`
4. Alur, R., Holzmann, G., Peled, D.: An analyzer for message sequence charts. Software - Concepts and Tools 17(2), 70–77 (1996)
5. Ardis, M.A.: Formal Methods for Telecommunication System Requirements: A Survey of Standardised Languages. Annals of Software Engineering 3 (1997)
6. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008)
7. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
8. Billington, J., Gallasch, G., Kristensen, L.M., Mailund, T.: Exploiting Equivalence Reduction and the Sweep-Line Method for Detecting Terminal States. IEEE Transactions on Systems, Man, and Cybernetics. Part A: Systems and Humans 34(1), 23–38 (2004)
9. Billington, J., Gallasch, G.E., Han, B.: A Coloured Petri Net Approach to Protocol Verification. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 210–290. Springer, Heidelberg (2004)
10. Billington, J., Han, B.: Modelling and Analysing the Functional Behaviour of TCPs Connection Management Procedures. International Journal on Software Tools for Technology Transfer 9(3-4), 269–304 (2007)
11. Billington, J., Vanit-Anunchai, S.: Coloured Petri Net Modelling of an Evolving Internet Protocol Standard: The Datagram Congestion Control Protocol. Fundamenta Informaticae 88(3), 357–385 (2008)
12. Billington, J., Yuan, C.: On Modelling and Analysing the Dynamic MANET On-Demand (DYMO) Routing Protocol. In: Jensen, K., Billington, J., Koutny, M. (eds.) Transactions on Petri Nets and Other Models of Concurrency III. LNCS, vol. 5800, pp. 98–126. Springer, Heidelberg (2009)

13. Bochmann, G.: Finite state description of protocols. Computer Networks, 361–372 (1978)
14. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks 14, 25–59 (1987)
15. Chakeres, I.D., Perkins, C.E.: Dynamic MANET On-demand (DYMO) Routing. Internet-Draft. Work in Progress (July 2007), http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-10.txt
16. Chakeres, I.D., Perkins, C.E.: Dynamic MANET On-demand (DYMO) Routing. Internet-Draft. Work in Progress (November 2007), http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-11.txt
17. Choppy, C., Dedova, A., Evangelista, S., Hong, S., Klai, K., Petrucci, L.: The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 145–164. Springer, Heidelberg (2010)
18. Choppy, C., Petrucci, L., Reggio, G.: A Modelling Approach with Coloured Petri Nets. In: Kordon, F., Vardanega, T. (eds.) Ada-Europe 2008. LNCS, vol. 5026, pp. 73–86. Springer, Heidelberg (2008)
19. Christensen, S., Kristensen, L.M., Mailund, T.: A Sweep-Line Method for State Space Exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
20. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting Symmetries in Temporal Logic Model Checking. Formal Methods in System Design 9, 77–104 (1996)
21. Comer, D.E.: Internetworking with TCP/IP vol. 1: Principles, Protocols, and Architecture, 5th edn. Prentice-Hall (2005)
22. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPV6) Specification. RFC 2460 (December 1998)
23. Ding, L.G., Liu, L.: Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 132–151. Springer, Heidelberg (2008)
24. Emerson, E.A., Sistla, A.P.: Symmetry and Model Checking. Formal Methods in System Design 9, 105–131 (1996)
25. Espensen, K.L., Kjeldsen, M.K., Kristensen, L.M.: Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 152–170. Springer, Heidelberg (2008)
26. ETSI. ETSI ES 201 873-1: Methods for Testing and Specification; The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language
27. Evangelista, S., Westergaard, M., Kristensen, L.M.: The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. Transactions on Petri Nets and Other Models of Concurrency 3, 189–215 (2009)
28. Examples of Industrial Use of CPNs, http://cs.au.dk/cpnets/industrial-use/
29. Fehnker, A., van Glabbeek, R., Hofner, P., McIver, A., Portmann, M., Tan, W.: Modelling and Analysis of AODV in UPPAAL. In: Proc. of 1st Workshop on Rigorous Protocol Engineering (2011)
30. Fleischer, P., Kristensen, L.M.: Modelling and Validation of Secure Connection Establishment in a Generic Access Network Scenario. Fundamenta Informaticae 94(3-4), 361–386 (2009)
31. Gallasch, G.E., Billington, J.: Using Parametric Automata for the Verification of the Stop-and-Wait Class of Protocols. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 457–473. Springer, Heidelberg (2005)

32. Gallasch, G.E., Billington, J.: A Parametric State Space for the Analysis of the Infinite Class of Stop-and-Wait Protocols. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 201–218. Springer, Heidelberg (2006)
33. Gallasch, G.E., Billington, J., Vanit-Anunchai, S., Kristensen, L.M.: Checking Safety Properties On-the-fly with the Sweep-Line Method. International Journal on Software Tools for Technology Transfer (STTT) 9(3-4), 371–392 (2007)
34. Gallasch, G.E., Ouyang, C., Billington, J., Kristensen, L.M.: Experimenting with Progress Mappings for the Sweep-Line Analysis of the Internet Open Trading Protocol. In: Proc. of 5th Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools (CPN 2004), pp. 19–38 (2004)
35. Gallasch, G.E., Han, B., Billington, J.: Sweep-Line Analysis of TCP Connection Management. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 156–172. Springer, Heidelberg (2005)
36. Gehlot, V., Hayrapetyan, A.: A Formalized and Validated Executable Model of the SIP-based Presence Protocol for Mobile Applications. In: Proceedings of the 45th Annual ACM Southeast Regional Conference, pp. 185–190. ACM (2007)
37. Gehlot, V., Hayrapetyan, A.: A CPN Model of a SIP-Based Dynamic Discovery Protocol for Webservices in a Mobile Environment. In: Proc. of 7th Workshop and Tutorial onPractical Use of Coloured Petri Nets and the CPN Tools, CPN 2006 (2006)
38. Genest, B., Muscholl, A., Peled, D.: Message sequence charts. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 537–558. Springer, Heidelberg (2004)
39. Gordon, S.: Formal Analysis of PANA Authentication and Authorisation Protocol. In: Proc. of 9th International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 277–284. IEEE Computer Society (2008)
40. Gordon, S., Billington, J.: Analysing the WAP Class 2 Wireless Transaction Protocol Using Coloured Petri Nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 207–226. Springer, Heidelberg (2000)
41. Gordon, S., Kristensen, L.M., Billington, J.: Verification of a Revised WAP Wireless Transaction Protocol. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 182–202. Springer, Heidelberg (2002)
42. Grimstrup, P.: Interworking Description for IKEv2 Library. In: Ericsson Internal. Document No. 155 10-FCP 101 4328 Uen (September 2006)
43. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8(3), 231–274 (1987)
44. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
45. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley (2004)
46. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice-Hall (1991)
47. The Internet Engineering Task Force (IETF), `http://www.ietf.org`
48. Ip, C.N., Dill, D.L.: Better Verification Through Symmetry. Formal Methods in System Design 9, 41–75 (1996)
49. ISO9074. Information Processing Systems - Open Systems Interconnection: ESTELLE (FOrmal Description Technique Based on an Extended State Transition Model)
50. ISO89 ISO/IEC. Information Processing Systems - Open Systems Interconnection: LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807 (February 1989)

51. ITU-T. Z.120: Message Sequence Charts (MSC) (1996)
52. ITU-T. Z.109: SDL-2000 Combined with UML (2000)
53. ITU-T. X.680 to X.683: Abstract Syntax Notation One (2002)
54. ITU-T. X.692 - Encoding Control Notation (2002)
55. ITU-T. Z.100-Z.106: Specification and Description Language (SDL) (2010)
56. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. vol. 1: Basic Concepts. Monographs in Theoretical Computer Science. Springer (1992)
57. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Analysis Methods. Monographs in Theoretical Computer Science, vol. 2. Springer (1994)
58. Jensen, K.: Condensed State Spaces for Symmetrical Coloured Petri Nets. Formal Methods in System Design 9, 7–40 (1996)
59. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
60. Jensen, K., Kristensen, L.M., Mailund, T.: The sweep-line state space exploration method. Theoretical Computer Science 429, 169–179 (2012)
61. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer (STTT) 9(3-4), 213–254 (2007)
62. Jørgensen, J.B., Kristensen, L.M.: Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. IEEE Transactions on Parallel and Distributed Systems 10(7), 714–732 (1999)
63. Jørgensen, J.B., Kristensen, L.M.: Verification of Coloured Petri Nets Using State Spaces with Equivalence Classes. In: Petri Net Approaches for Modelling and Validation, Lincoln Europa. LINCOM Studies in Computer Science, ch. 2, vol. 1, pp. 17–34 (2003)
64. Kaufman, C.: Internet Key Exchange Protocol. RFC 4306 (December 2005)
65. Kent, S., Seo, K.: Security Architecture for the Internet Protocol. RFC 4301 (December 2005)
66. Kristensen, L.M.: A Perspective on Explicit State Space Exploration of Coloured Petri Nets: Past, Present, and Future. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 39–42. Springer, Heidelberg (2010)
67. Kristensen, L.M., Jensen, K.: Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 248–269. Springer, Heidelberg (2004)
68. Kristensen, L.M., Mailund, T.: A Generalised Sweep-Line Method for Safety Properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002)
69. Kristensen, L.M., Mailund, T.: Efficient Path Finding with the Sweep-Line Method Using External Storage. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 319–337. Springer, Heidelberg (2003)
70. Kristensen, L.M., Mechlenborg, P., Zhang, L., Mitchell, B., Gallasch, G.E.: Model-based Development of COAST. STTT 10(1), 5–14 (2007)
71. Kristensen, L.M., Jørgensen, J.B., Jensen, K.: Application of Coloured Petri Nets in System Development. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN. LNCS, vol. 3098, pp. 626–685. Springer, Heidelberg (2004)

72. Kristensen, L.M., Valmari, A.: Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 104–123. Springer, Heidelberg (1998)
73. Kristensen, L.M., Westergaard, M.: Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 215–230. Springer, Heidelberg (2010)
74. Kristensen, L.M., Westergaard, M., Nørgaard, P.C.: Model-Based Prototyping of an Interoperability Protocol for Mobile Ad-Hoc Networks. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 266–286. Springer, Heidelberg (2005)
75. Lai, R., Jirachiefpattana, A.: Communication Protocol Specification and Verification. Kluwer Academic Publishers (1998)
76. Lakos, C.: Modelling Mobile IP with Mobile Petri Nets. Transactions on Petri Nets and Other Models of Concurrency 5800, 127–158 (2009)
77. Liu, L.: Verification of the SIP Transaction Using Coloured Petri Nets. In: Mans, B. (ed.) Thirty-Second Australasian Computer Science Conference (ACSC 2009), Wellington, New Zealand. CRPIT, vol. 91, pp. 63–72. ACS (2009)
78. Liu, L.: Security Analysis of Session Initiation Protocol - A Methodology Based on Coloured Petri Nets. In: Proc. of the 2010 International Cyber Resilience Conference (2010)
79. Liu, L., Billington, J.: Verification of the Capability Exchange Signalling protocol. STTT 9(3-4), 305–326 (2007)
80. Liu, M.T.: Protocol Engineering. Advances in Computers 29, 79–195 (1989)
81. Lorentsen, L., Kristensen, L.M.: Modelling and Analysis of a Danfoss Flowmeter System Using Coloured Petri Nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 346–366. Springer, Heidelberg (2000)
82. IETF Mobile Ad-hoc Networks Discussion Archive, `http://www1.ietf.org/mail-archive/web/manet/current/index.html`
83. Mailund, T.: Analysing infinite-state systems by combining equivalence reduction and the sweep-line method. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 314–333. Springer, Heidelberg (2002)
84. Malik, R., Mühlfeld, R.: A case study in verification of uml statecharts: the profisafe protocol. Universal Computer Science 9(2), 138–151 (2003)
85. Malkin, G.: RIP Version 2. RFC 4822 (February 2007)
86. Milner, R.: Communication and Concurrency. Prentice-Hall International (1989)
87. Mortensen, K.H.: Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 367–386. Springer, Heidelberg (2000)
88. Narten, T., Nordmark, E., Simpson, W.: Neighbor Discovery for IP Version 6 (IPv6), RFC 2461 (December 1998)
89. Ouyang, C., Billington, J.: On Verifying the Internet Open Trading Protocol. In: Bauknecht, K., Tjoa, A.M., Quirchmayr, G. (eds.) EC-Web 2003. LNCS, vol. 2738, pp. 292–302. Springer, Heidelberg (2003)
90. Ouyang, C., Billington, J.: Formal Analysis of the Internet Open Trading Protocol. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, pp. 1–15. Springer, Heidelberg (2004)

91. Ouyang, C., Kristensen, L.M., Billington, J.: A Formal and Executable Specification of the Internet Open Trading Protocol. In: Bauknecht, K., Tjoa, A.M., Quirchmayr, G. (eds.) EC-Web 2002. LNCS, vol. 2455, pp. 377–387. Springer, Heidelberg (2002)
92. Ouyang, C., Kristensen, L.M., Billington, J.: A Formal Service Specification for the Internet Open Trading Protocol. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 352–373. Springer, Heidelberg (2002)
93. Petri, C.A.: Kommunikation mit Automaten. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2 (1962)
94. Popovic, M.: Communication Protocol Engineering. CRC Press (2006)
95. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 450–462. Springer, Heidelberg (2003), http://www.cpntools.org
96. Ravn, A.P., Srba, J., Vighio, S.: Modelling and verification of web services business activity protocol. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 357–371. Springer, Heidelberg (2011)
97. Reisig, W.: Petri Nets - An Introduction. EATCS Monographs on Theoretical Computer Science, vol. 4. Springer (1985)
98. Stern, U., Dill, D.L.: Improved Probabilistic Verification by Hash Compaction. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995)
99. Suriadi, S., Ouyang, C., Smith, J., Foo, E.: Modeling and Verification of Privacy Enhancing Protocols. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 127–146. Springer, Heidelberg (2009)
100. Ullman, J.D.: Elements of ML Programming. Prentice-Hall (1998)
101. Valmari, A.: A Stubborn Attack on State Explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
102. Valmari, A.: Stubborn Sets of Coloured Petri Nets. In: Proc. of ICATPN 1991, pp. 102–121 (1991)
103. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
104. Vanit-Anunchai, S.: Towards Formal Modelling and Analysis of SCTP Connection Management. In: Proc. of CPN 2009, pp. 163–182 (2008)
105. Vanit-Anunchai, S., Billington, J., Gallasch, G.E.: Analysis of the Datagram Congestion Control Protocols Connection Management Procedures Using the Sweep-line Method. International Journal on Software Tools for Technology Transfer 10(1), 29–56 (2008)
106. Villapol, M.E., Billington, J.: A Coloured Petri Net Approach to Formalising and Analysing the Resource Reservation Protocol. CLEI Electron. J. 6(1) (2003)
107. Villapol, M.E., Billington, J.: Analysing Properties of the Resource Reservation Protocol. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 377–396. Springer, Heidelberg (2003)
108. Vixie, P.: Dynamic Updates in the Domain Name System. RFC 2136 (April 1997)
109. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An Extensible Platform for State Space Analysis. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 303–312. Springer, Heidelberg (2009), http://www.daimi.au.dk/~ascoveco/download.html

110. Westergaard, M., Kristensen, L.M., Brodal, G.S., Arge, L.A.: The ComBack Method – Extending Hash Compaction with Backtracking. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 445–464. Springer, Heidelberg (2007)
111. Westergaard, M., Lassen, K.B.: The BRITNeY Suite Animation Tool. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 431–440. Springer, Heidelberg (2006)
112. Westergaard, M.: A Game-theoretic Approach to Behavioural Visualisation. Electr. Notes Theor. Comput. Sci. 208, 113–129 (2008)
113. Wolper, P., Leroy, D.: Reliable Hashing without Collision Detection. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)

Business Process Modeling Using Petri Nets

Kees M. van Hee, Natalia Sidorova, and Jan Martijn van der Werf*

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{k.m.v.hee,n.sidorova,j.m.e.m.v.d.werf}@tue.nl

Abstract. Business process modeling has become a standard activity in many organizations. We start with going back into the history and explain why this activity appeared and became of such importance for organizations to achieve their business targets. We discuss the context in which business process modeling takes place and give a comprehensive overview of the techniques used in modeling. We consider bottom up and top down approaches to modeling, also in the context of developing correct-by-construction models of business processes. The correctness property we focus on is soundness, or weak termination, basically meaning that at every moment of its execution, a process has an option to continue along an execution path leading to termination, which is an important sanity check for business processes. Finally, we discuss analogies between business processes and software services and their orchestrations and argue the applicability of the described modeling techniques to the world of services.

1 Introduction

The concept of a *business process* (BP) is as old as humanity. A BP is the set of interdependent tasks and resources needed to produce some service or product. On top of this set there are constraints or business rules that have to be met. Business processes form the heart of organizations: they should make possible that organizations can realize their goals. Although business processes always have existed, the description, or modeling, of BPs started only recently. In the twentieth century auditors and accountants started working on BP specifications in their field and later it became a hot topic in quality management. In these early days process descriptions were used for documentation, and for facilitating communication between persons. As a result, these descriptions were informal, often in a natural language enriched with some diagrams.

Around the year 1990 business processes became a hot topic in industry, as people became aware of the fact that we were not fully exploiting the power of computer systems. Information systems supporting business processes were

* Supported by the PoSecCo project (project no. 257129), partially co-funded by the European Union under the Information and Communication Technologies (ICT) theme of the 7th Framework Programme for R&D (FP7).

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 116–161, 2013.

data-oriented, meaning they were targeted in recording the status of objects that played a role in business processes, in a database. Consequently, database technology was the main technology at the beginning of the nineties. Moreover, information systems were built to support existing business processes by automating their tasks by imitation of the human work by a computer. The papers and books of Hammer and Champy [32, 33] gave very convincing examples of inefficient use of computers and they advocated the re-engineering of business processes before the development of supporting information systems. This was also the first time that the term "engineering" was applied to business processes. Indeed computers can process information in completely different ways from people and it is logical to exploit these possibilities to make business processes more effective and more efficient.

The awareness of the importance of business processes has triggered the introduction of the concept of *process-aware information systems*. Information systems became more than recorders of the status of objects—they started to also focus on business-relevant *events*. The information systems became proactive in the sense that they, for instance, started to control the right order of task execution, keep track of deadlines and distribute the work between resources.

The most notable implementations of the concept of process-aware information systems are *workflow management systems*, a class of generic components for the construction of information systems. Workflow management systems have become the counterparts of database management systems. While a database management system is configured by a database schema and a set of constraints, a workflow management system is configured with a process model. A workflow engine can be embedded in a larger information system the same way as database engines can. From ca. 1995 up to around 2005, there was a strong focus on the support of single business processes with workflow engines. After that, the interest shifted to *cooperating business processes*, as encountered in supply chains and in BP outsourcing.

One of the most recent forms of composition trend in the last decade is the Service Oriented Computing (SOC) [14, 68]. In this paradigm for systems development, closely related to the paradigm of Service Oriented Architecture (SOA) [18, 68], systems are considered as components that deliver services to each other, like businesses in a supply chain. Each component runs a process, to orchestrate the service and in fact such a processes can be seen as a business process. The business processes in the real world are in a way mirrored in the components of the information systems. So here we also encounter the cooperation of business processes, which generates new scientific challenges to develop correct working systems.

The focus of this article is to give insight in the role of business processes, the modeling of business processes and the use of these models in BP management. Instead of presenting new theoretical results we try to give insight as well as an overview of theoretical results. In Section 2 we study the context of business processes, i.e. the world in which business processes play their role. Then in Section 3, we study the modeling of business processes, in particular a bottom

up and a top down approach, and we will analyse the correctness of BP models. We only focus on a generic property: weak termination, which is the ability of always being able to reach a final state. This turns out to be a very important sanity check for business processes. We focus on correctness by construction principles. Finally in Section 4, we move from business processes to services and in particular to computerized services as we can find them in modern web-based information systems. Services incorporate or emulate business processes. Services cooperate with each other and so we have to study the cooperation of two or more business processes which provides new challenges. We conclude the article in Section 5.

2 Context

We start with the informal introduction of a set of related concepts from the world in which BPs play their roles. Many of these concepts will be formalized or modeled in the next section. However modeling always has to serve some purpose. In order to do so, we first need to understand the context of the part of the world we are modeling. This means that we have to classify the real world entities, either abstract or physical, and map them to the concepts; see also [5].

2.1 Business Process Concepts

An *organization* is a system consisting of humans, machines, materials, buildings, data, knowledge, rules and other means, with a set of *goals* to be met. Typical examples of organizations are companies, factories, hospitals, schools and governmental institutions. We also consider *business units* or departments within a larger organization as organizations, and similarly, we also consider two or more cooperating organizations as one organization. Most organizations have, as one of their main goals, the *creation* or *delivery* of (physical) *products* or (abstract) *services*.

The creation of services and products is performed in *business processes* (BP). A BP is a set of *tasks* with *causal dependencies* between tasks. The five basic *task ordering principles* are

- *Sequence* pattern: putting tasks in a linear order;
- *Or-split* pattern: selecting one branch to execute;
- *And-split* patterns: all branches will be executed;
- *Or-join* patterns: one of the incoming branches should be ready in order to continue; and
- *And-join* pattern: all incoming branches should be ready in order to continue.

As will be shown in the next section, these basic patterns are closely related to the well-known four *control flow* constructs from programming:

- *Sequence* construct;
- *Choice* construct;

– *Iteration* constructs, like "repeat until" and "while do"; and
– *Parallel* construct.

Actually we can make the above programming constructs with the five task ordering patterns.

For the execution of tasks *resources* are required. Resources can either be *durable* or *consumable*. The first kind is available again after execution of one or more tasks, like a catalyst in a chemical process. Typical examples of this kind are humans, machines, computer systems, tools, information and knowledge. Consumable resources disappear during the task execution. Examples are energy, money, materials, components and data. The results or output of a task can be considered as resources for subsequent tasks or as final products or services. Two kinds of durable resources are of particular importance: the humans as a resource, called *human resources*, and information and knowledge, which we will call *data resources*. Since human activities are sometimes replaced by computer systems, we use the term *agents* as a generic term for human and system resources.

A BP is an abstract notion: no physical object exists that can be identified as a BP. A BP has many different forms of appearance and different names, depending on the context. For example, an administrative *procedure* for handling claims in an insurance company is a BP, and a medical protocol to treat a form of cancer describes another BP. Even a *recipe* for cooking or an *algorithm* for computing can be seen as BPs. Thus, we consider any form of task structuring in order to create a product or a service as a BP.

Tasks are viewed as *atomic* pieces of work. Atomic means that the task is executed without breaks and that the agents and durable resources needed for the execution are available at the beginning and are released at the end. We do not consider the content of a task. Later we will see that task atomicity is a view or a modeling decision which can be adjusted at a later stage of modeling: a task can be substituted by another process, which becomes a *subprocess* of the original process.

An important feature of a BP is that it can be repeated, meaning a BP has multiple *instances*. An instance can be seen from two different view points: (1) the entity that is in progress of being created, called a *case*, and (2) the process of creation which can be seen as a *project*. We will use the term case for both views. A case is at each moment in time in some *state*. In the initial state the process is ready to start and in the final state the product or service is completed. The state of a case is determined by (1) *case conditions*, that are true or false for the case at that state and (2) *case variables*, data connected to the case. There are two kinds of case variables: *routing parameters* used for routing through the process and a *case document*, which is a file with all relevant data of the case.

It is possible to have parallel task executions for a case, which means that *concurrently* different resources can be busy with different tasks for the same case. Besides the concurrency within a single case, BPs can handle several cases concurrently. However if two or more cases are executed concurrently, we assume they are *independent* of each other: they do not influence each others final result.

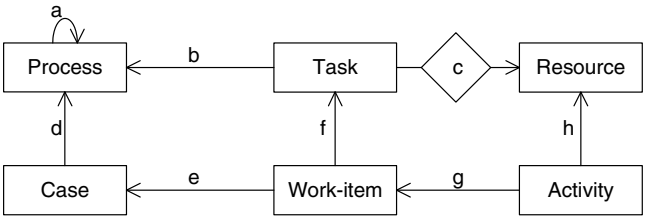


Fig. 1. Business process concepts and their relations

Cases can share resources, which may influence the *scheduling* of tasks, respecting the task ordering requirements of the BP. Sharing of resources is often a form of competition between the cases. In this context, it is useful to distinguish the notions of a *work item* and an *activity*: a work item is the combination of a task and a case, while an activity is the combination of a work item and resources. A work item can be seen as a task instance while an activity is the execution of a work item. Figure 1 shows the concepts in a BP and their relationship.

Besides the causal dependencies expressed by the task ordering principles, there are often many more *constraints* on the execution of BPs. These constraints are called *business rules* and there are special languages to express them (cf. [9,36,67]). An example of a business rule is the four-eyes-principle that says that certain tasks in one case should not be executed by the same human resource. Such rules cannot be expressed by task ordering principles and have to be guaranteed by using other means. It is of course very important to be able to verify if certain BP satisfies a set of business rules.

BPs can be classified according to their function within the organization:

- *Primary* processes: they are dedicated to the primary goals of the organization, namely the creation of the products and services using resources;
- *Secondary* processes, also called *supporting* or *enabling* processes: they are dedicated to providing and maintaining the resources for the primary processes as well as maintaining relationships with the customers and suppliers; and
- *Tertiary* processes, also called *management* processes: they are dedicated to the *control* of the organization as a whole and the *coordination* of the primary and secondary processes.

Here we encounter again a dilemma: the chosen perspective determines the classification. For example, when a business unit of a department performs a supporting process for the department, this supporting process can be a primary process for the business unit. Similarly, a tertiary process of an organization can be a primary process of a business unit. On the one hand this might be confusing, on the other hand, it gives an opportunity to define clear goals, customers and suppliers for all BPs. A typical example concerns *authorization* of human resources, i.e. the process of delegation of rights to perform functions. On the one hand this can be considered as a kind of special task within a primary process

itself but it can also be seen as a secondary process, as it concerns (human) resource management.

A subclass of the tertiary processes consists of so-called *inter-organizational* BPs. These processes exchange information, resources and cases between different, but *cooperating* organizations. In one organization the control can be hierarchical, which is normally not the case in cooperating independent organizations. There, coordination is the key activity. Examples of inter-organizational BPs occur in *supply chains*, where the BPs of the participating organizations work together as co-makers. Another example is an electronic market place where the coordination between supply and demand is organized by means of market mechanisms. Here we enter the field of *communicating* BPs which is the topic of Section 4.

2.2 Supporting Information Systems

Process aware information systems support organizations by means of three basic functions: *monitoring*, *planning* and *execution* of tasks in BPs. The monitoring concerns the recording of all *events* in the processes. This also enables providing management information: not only the status of each of the cases, like running or completed, but also aggregated information like the total number of cases running currently, average processing time and waiting time of all cases.

The planning function of a process aware information system is more proactive, as it concerns the selection of tasks that are ready to be executed, the allocation of resources for a task and preparation of the execution by transferring the right data resources to the agents (human resources or computer systems). More and more tasks of BPs are executed autonomously by information systems. Specifically in financial and governmental institutions where BPs consist of information processing only, e.g. the transfer of money, granting a mortgage or the registration of a marriage. This kind of BPs only require information processing. Sometimes human judgement is essential, but in many situations there are formalized rules for making decisions automatically.

In principle active databases (cf [69]) support these three functions. However, the set of rules involved becomes very complex, so that people lose overview, with an inconsistent set of rules as a consequence. This led to the development of a new class of software components: *workflow management systems* [91], as a counterpart of database management systems. Workflow management systems are configured by means of a *process model* like a database management system is configured by a database schema or an entity-relationship diagram. The term "workflow" is often considered to be a synonym for "BP".

An important component within a workflow management system is the *workflow engine* which takes care of the distribution of tasks over the agents. This provides flexibility: if a BP has to be changed, only the process model has to be adapted, without changing the rest of the information system. Another part of the workflow management system concerns the authorization of human resources. This function is similar to the authorization of users of databases. The support for handling data resources is very limited in workflow management

systems. Only some case parameters are considered and in particular used for routing decisions. This has been a conscious design choice: workflow management systems should only be concerned with the distribution of work and not with the content. There is another class of information systems, called *case handling systems* [13, 37], that combine workflow management with data handling. There exists a standard architecture for worfklow management systems (cf [91]). However most workflow management systems have their own process modeling language, although there are some standards as well [66, 92].

In many organizations information systems are realized by means of *standard application packages* like ERP-systems (Enterprise Resource Planning systems). These systems are actually *configurable information systems*. In the past they had support for a predefined set of possible BPs. Today there is a trend to migrate from monolithic systems with a finite set of predefined options, to more flexible component-based systems with one or more workflow engines inside.

Another, but related, trend in the world of information systems is called Service Oriented Architecture (SOA). According to this paradigm, information systems are build as *loosely coupled components* that deliver *services*, using other services. The "loose coupling" has three characteristics: (1) components communicate *asynchronously* by exchanging messages via *ports*, (2) only the port *protocols* have to be known in order to couple components and (3) the coupling can be done at runtime via a *service broker*, which is in fact dynamic binding of services. Components within such a system have internally a process that defines the service, called the *orchestration* of the service. An orchestration process is actually a workflow and each delivery of a service is a case. For instance, the Web Service Business Process Execution Language (WS-BPEL, [15]) is a language belonging to the web technology family, to orchestrate components.

The cooperation between components is sometimes called the *choreography*. There is a great analogy between BPs and service in a SOA. First, many BPs are supported by or even replaced by a service component. Specifically in retail, financial institutions and government we see that BPs involving the customers are replaced by web services where the customer is in direct contact with the information system of the organization without interference of employees unless an exception occurs. In those cases, the original BPs are in a way simulated by the service components. Secondly, service components in a SOA behave more or less like independent business units in one organization. In particular the tendency in organizations to outsource non-key sub-processes is mirrored in the execution of a service by calling other service components for particular tasks.

2.3 Role of Models

Organizations are not isolated: they create products or deliver services to their *customers* and obtain the resources needed from *suppliers*. Both suppliers and consumers can be persons or organizations themselves. All persons or organizations that have some interest in an organization are called *stakeholders*. Beside customers and suppliers, also employees (human resources), management, share holders and in many cases the government (e.g., the tax department) are

stakeholders. Stakeholders have different objectives within an organization. Therefore, in modeling BPs, it is essential to know in advance for which stakeholders a problem is being solved.

Modeling a system is a way of *understanding* the system. There is empirical evidence that making a precise model of a complex entity, such as a BP, reveals all kind of details that we are overlooking otherwise. Formalizing an informal description and then constructing a new informal description from the formalization results in a better description. There are plenty of reasons to *document* a BP. For example as teaching material for new employees, for quality control and for the development of supporting information systems. We distinguish two kind of models: *descriptive models*, also called "as is" models, that describe how a system or processes actually is, and *normative models*, also called "to be" models, that describe how the process should work. In fact, models exist for many reasons:

1. to understand processes
2. to document processes
3. to analyse processes
4. to monitor and audit processes
5. to improve or optimize processes
6. to outsource processes
7. to construct or redesign processes
8. to execute processes

Models help to analyze processes. There are two kinds of *analysis* of BPs:

- to verify *conformance*, i.e. to check if a BP satisfies a set of business rules (cf [36]);
- to determine the *performance*, i.e. to compute performance characteristics, like the time or the number of resources needed to reach a certain state.

For conformance analysis we use general purpose model checking techniques (e.g. [50, 89]) or dedicated techniques like structural analysis. Business rules typically concern the ordering of tasks and the use of resources and combinations of these. For performance analysis, simulation is the most used technique: the process model is used as simulation model (e.g. [39, 72]).

Monitoring a process is recording an execution path, which may include several cases of a BP in order to be able to reconstruct parts of the process. The most important function of monitoring is to produce reports of the past, either periodically, event-driven or just on demand. *Auditing* a process is checking if an execution path of a BP satisfies a set of business rules. Whereas conformance is checking the whole process, auditing is checking only the execution paths of the process. Of course if we are sure that a certain process is executed, then it is sufficient to check if all the business rules are satisfied, but if we are not sure a "to be" BP is really executed, then we have to audit the runtime system.

Based on performance analysis one may try to *improve* the performance of a BP by restructuring it or by reallocating resources. *Optimization* is improving

until some optimality criterion is reached. In practice often the term optimization is used in cases where it actually refers to improvement. The choice of a good optimality criterion is far from easy and since stakeholders do not know the optimal performance they are already satisfied if the performance has improved significantly.

Outsourcing is a kind of restructuring a BP by isolating a subprocess and replacing it by a process of another organization, called the supplier. This way, organizations can put their efforts in their core business, while other organizations can better perform in some tasks or sub-processes, because they have specialized skills or they can exploit economy or scale effects. By outsourcing, a particular kind of inter-organizational BP is created: the original BP is communicating with the subprocess of the supplier. This way, a supply chain is formed.

To *construct* a new BP or to *redesigning* an existing BP, one first makes a model of the process. Then we analyse this model to verify conformance and performance properties. After this, we create or implement the BP. Most of the time, implementation of a new or changed BP within an organization involves more details than the model expresses, like the location(s) where the process will be executed, the durable resource to be used and training human resources. In many cases implementation also involves the development of supporting information systems. However, the model plays an essential role in these activities.

In the *execution* of a BP the model is used to determine the *order of tasks*, the *authorization* of human resources, and the *allocation* of resources. Either this is done by a supporting information system or by human activities. In the first case the process model is used as a configuration parameter of the workflow management system, in the second case as manual or handbook.

2.4 Modeling Languages and Tools

In the nineties, various industries developed different workflow management systems. As a workflow management system needs a process model as configuration parameter, industry created many different languages to model BP. Most of these languages have a graphical syntax. Although it looks easy to define such a language, it turns out to be a difficult job, because of the dynamical semantics of processes.

Before the nineties there were already process modeling languages, mostly without formal semantics, like DFD (dataflow diagrams diagrams, cf [82]). However, these were not used for modeling of BPs. The most important examples of industrial process formalisms are: EPCs (Event-driven Process Chains) (cf [53]) with supporting tool ARIS (cf [75]), BPEL (cf [15]), BPMN (cf [66]), and UML-Activity Diagrams (cf [31]). The last three are industry standards. They all have still flaws in they formal semantics, although new releases become better. BPEL has no graphical representation.

Many of the industrial languages do not have formal semantics. Some have only an operational semantics in the form of a simulation tool that can simulate the behavior of a given model. If a formal semantics exists, it is mostly in terms of a (labeled) transition system or a (labeled) Petri net. Labeled transition

systems provide *interleaving semantics*, which means that concurrency of tasks is expressed as different possible orderings. Petri nets allow for *partial order semantics*. Formal semantics are essential for analysis purposes.

There were already very good process formalisms available from academia: the many types of Petri nets (cf [27, 70, 73]) and process algebras, e.g. CSP [48] and CCS [62]. However these formalisms are general purpose process formalisms and not tuned to BP, which means that direct support for some useful constructs like an implicit Or-split and Or-join is missing. Process algebras have no graphical syntax, and are difficult to understand by most practitioners. Petri nets, with its diagram technique is a much better candidate and indeed some workflow managements system (e.g. COSA [78] and YAWL [49]) are based on Petri nets. There is also an ISO standard for Petri nets [47]. A special class of Petri nets was developed, called workflow nets [1] and they are used frequently. The semantics of UML-Activity Diagrams and BPMN are converging to Petri nets. In this article we will use workflow nets for modeling BPs.

Many different *software tools* exist to support the modeling process. So we have *editors* to develop and publish models (cf [23, 30, 39, 57, 72, 75]), and tools to analyze models, either by model checking, structural analysis (e.g. [4, 50, 76, 87]) or simulation. Most modeling tools have simulation facilities (cf [39, 72]).

3 Modeling Business Processes

In the remainder we only consider Petri nets and in particular the class of workflow nets, to model BPs. For many purposes it is sufficient to consider classical Petri nets, i.e. with "black" tokens. However sometimes it is important to have the modeling power of colored Petri nets. For a formal definition of classical Petri nets as used in this paper see Appendix A. For more information about classical Petri nets, we refer the user to e.g. [73, 86]. For a formal definition of colored Petri nets see [51]. Here we give an informal introduction only.

A *classical Petri net* is a triple (P, T, F) where P is the set of *places*, T the set of *transitions* and F a function, called the *flow function*, that assigns to each pair of nodes of $(P \times T) \cup (T \times P)$ a natural number, possibly zero. There is a graphical notation for Petri nets where places are displayed as circles and transitions as rectangles. If $F(x, y) > 0$ for a pair of nodes (x, y) then we say that there is a directed arc from x to y and $F(x, y)$ is the arc weight. We say that x is an input node for y and that y is an output node for x.

Places may contain tokens. A distribution of tokens over places is called a *state*. In classical nets the state can be expressed as a function that assigns to each place the number of tokens in that place; this function is called a *marking* of the net. A transition is *enabled* if for all input places the number of tokens in each input place is at least the arc weight. An enabled transition can fire. If it fires, the marking changes: The number of tokens consumed from the input places of the transition and the number of tokens produced to the output places is defined by the weights of the corresponding arcs. Thus, when a transition fires, the Petri net "moves" from one state to another. In this way, a *labeled transition*

system is formed in which the Petri net transitions are the labels of the state transitions and the markings are the states. We call this the *reachability graph* of the Petri net. The combination of a Petri net and an initial marking is called a Petri system of system for short.

Since classical Petri nets are not Turing complete (cf. [16]), we often need more advanced features in modeling. For this purpose we introduce *inhibitor arcs* and *reset arcs*. To do so, we add for each of the arc types a new relation, I and H, respectively. The net depicted in Figure 2 has an inhibitor arc, denoted by an arc with a bullet head, between transition F and place p, indicating that transition F can only fire if place p is empty. The net has a reset arc, denoted by a dashed line, between transition G and place p, thus firing transition G empties place p.

In a *colored Petri net* (CPN) tokens have a value. Each place has an associated value type, also called a *color set*, and all tokens in a place have a value that belongs to that type. Each arc has one variable and transitions have a precondition, also called a *guard*, and a postcondition. A precondition is an expression with the variables of the input arcs of the transition as the only free variables. The post condition is an expression using the variables of the input arcs as well as of the output arcs. Figure 3 depicts a transition in a colored Petri net with its specification.

A transition in a colored Petri net is enabled if and only if for each input place a token can be found such that the precondition substituted with the actual values of the tokens is satisfied. An enabled transition may fire and if it does the post condition is evaluated with the input variables bound to the input tokens and then the output variables obtain a value. Tokens with these values are produced for each output place. In CPN tools (see [72]) and with CPNs as defined in [51] there is more freedom in modeling and more tokens can be consumed or produced for one place in one firing. In fact arcs may have expressions instead of only variables.

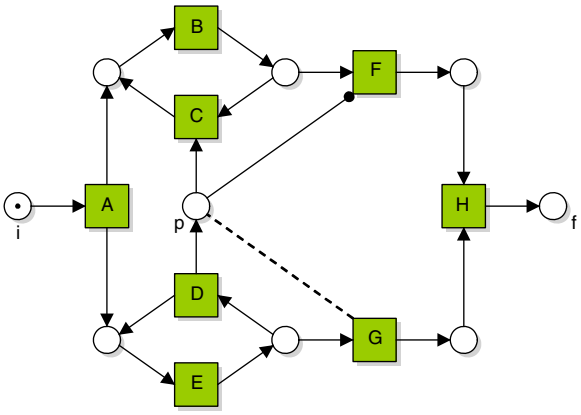


Fig. 2. Workflow net with reset arc and inhibitor arc

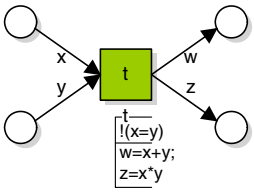


Fig. 3. Transition in a colored net

Token values are entities with one or more attributes, representing different properties. These attributes can be used to express arbitrary data. We identify two key attributes: *token identity* and *time stamps* that require a special treatment (see [35]). In case of token identity, each token carries and identifier. The precondition of each transition requires that all tokens consumed from the input places should have the same identifier [43, 74]. The tokens produced will obtain this same identifier.

The intuitive meaning of a time stamp of a token is the minimal time after which the token may be consumed by some transition. The firing rule for tokens with a time stamp is as follows. In order to determine which transition may fire next, all combinations of input tokens are considered for all transitions and a transition is selected for firing if its maximal time stamp of all its input tokens, is minimal over all enabled transitions. This time is called the *transition time* of the marking. We assume timed Petri nets to be *eager*, which means any transition fires as soon as possible. The tokens produced when a transition fires, obtain a time stamp which is the transition time plus some delay, depending on the inscription of the specific output arcs. Also for colored Petri nets we allow inhibitor arcs and reset arcs with the same semantics as for classical Petri nets.

In the remainder of this article, when we consider classical or colored Petri nets we will assume they do not have inhibitor or reset arcs, unless we explicitly require this.

3.1 Workflow Nets

For modeling convenience later, we start with a definition that generalizes the classical definition of a workflow net. A workflow net is a Petri net with two sets of special nodes: *initial* and *final* nodes. The first have no input nodes and the last have no output nodes. In fact we only use two types: one where all the special nodes are places and one where they all are transitions.

Definition 1 (Workflow net). *A* workflow net *N is a 5-tuple (P, T, F, E, C) where (P, T, F) is a Petri net, E is the set of* initial nodes *and C is the set of* final nodes *such that $E, C \subseteq P$ or $E, C \subseteq T$, ${}^\bullet E = C^\bullet = \emptyset$, and all nodes $(P \cup T)$ of (P, T, F) are on a directed path from a node in E to a node in C.*

- *If $E, C \subseteq P$ we say N is a* place-bordered WFN *(WFN-s), also called a* multi workflow net.

- *If $E, C \subseteq T$ we say N is a* transition-bordered WFN *(WFN-t).*
- *The* completion N^+ *of a WFN-s is a WFN-s (see Figure 4(a))*

$$N^+ = (\ P \cup \{i,f\},\ T\ \cup\ \{t_i, t_f\},\\ F \cup \{(i,t_i) \mapsto 1, (t_f,f) \mapsto 1\}\\ \quad \cup \{(t_i,e) \mapsto 1 \mid e \in E\} \cup \{(c,t_f) \mapsto 1 \mid c \in C\},\\ \{i\},\ \{f\}\ )$$

where $i, f \notin P$ and $t_i, t_f \notin T$ are fresh places and transitions, respectively.

- *The* completion N^+ *of a WFN-t is a WFN-s (see Figure 4(b))*

$$N^+ = (\ P \cup \{i,f\},\ T,\\ F \cup \{(i,e) \mapsto 1 \mid e \in E\} \cup \{(c,f) \mapsto 1 \mid c \in C\},\\ \{i\},\ \{f\}\ )$$

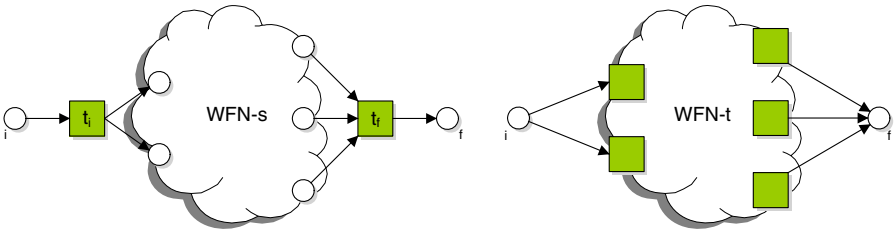
where $i, f \notin P$ are fresh places.

The completion transforms a WFN-s or a WFN-t into a WFN-s with a single input place and a single output place, as displayed in Figure 4. Such a WFN-s is a classical workflow net [1]. Actually the behavioral properties for WFN we will consider later are expressed in terms of the completions, i.e., in terms of classical workflow nets.

Definition 2 (Classical workflow net). *A WFN-s $N = (P, T, F, E, C)$ is a* classical workflow net *(WFN) iff $E = \{i\}$ and $C = \{f\}$ for some $i, f \in P$.*

- *Its workflow system is defined by $\mathcal{N} = (N, [i], \{[f]\})$.*
- *Its* closure *is defined by $\overline{N} = (P, T \cup \{t^*\}, F \cup \{((f,t^*),1),((t^*,i),1)\}$ where $t^* \notin T$ is a fresh transition.*
- *Its* fused closure *is defined by $N^* = (P \setminus \{f\}, T, (F \setminus {}^\bullet f \times \{f\}) \cup ({}^\bullet f \times \{i\}))$.*

As in classical Petri nets, we identify three important workflow net classes (cf [26]). These are: (1) free choice workflow nets (FC-WFN), where the underlying Petri nets is a free choice net, (2) state machines workflow nets (S-WFN), where the underlying Petri net is a state machine, also called S-net and marked graph workflow nets (T-WFN) where the underlying Petri net is a marked graph, also called T-net.



(a) WFN-s (b) WFN-t

Fig. 4. The completion of a WFN-s and WFN-t net

In order to model resources we extend the definition of workflow nets with resources. Resources are an additional set of places from which transitions can consume and produce tokens representing the resources they need or deliver.

Definition 3 (Resource-constrained workflow net [40]). *A* Resource-constrained WFN *(RCWFN) N is a 7-tuple $(P, R, T, F_c, F_r, E, C)$ such that*

- $P \cap R = \emptyset$*, P are called the* case *places and R the* resource *places;*
- $F_c : (P \times T) \cup (T \times P) \to \mathbb{N}$ *and* $F_r : (R \times T) \cup (T \times R) \to \mathbb{N}$*;*
- $(P \cup R, T, F_c \cup F_r)$ *is a Petri net;*
- (P, T, F_c, E, C) *is a WFN and this net is called the* production net *of N.*

So we have extended a WFN with an additional set of places, called *resource places*, and in that context we call the other places *case places*, and similarly we speak of resource tokens and case tokens.

3.2 Expressing Business Process Concepts

When modeling real life systems with Petri nets we always have to make a choice between two paradigms: either we model (time consuming) activities by transitions or by places. In the first choice the marking or state indicates a situation of rest and the transitions model the activities that may lead to a new state of rest. According to that paradigm, transitions are given names that reflect actions like, "move" or "print" while places have names that express a status, like "in stock" or "at home". In this way, transitions are named with a verb and places with a noun. According to the latter paradigm, transitions stand for instantaneous events and places may reflect a status as well as an activity. A place that represents an activity should have an input transition that represents the start event and an output transition that represents the stop event. In this section we will express the concepts of task, task ordering, case and resource in terms of workflow nets.

Tasks and Their Ordering. A *task* represents an activity and so it can be modeled in two ways: either a task is modeled as one transition or as a pair of transitions, where the first one is the *start event* of the task and the second one the *stop event* of the task (see Figure 5). Note that in classical Petri nets, time is not modeled, but only the order of execution can be expressed. In colored Petri nets time can be expressed using time stamps for tokens, which we will explain later. Although it is not necessary, it is a best practice to let a task have only

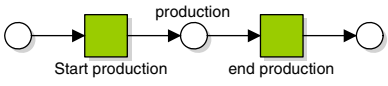


Fig. 5. Activity modeled with a start and stop event

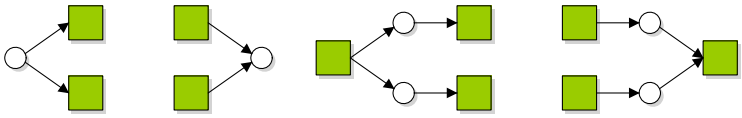
one input place and one output place. In case of start and stop events: the start event has one input and the stop event one output place.

The *ordering* of tasks, also called the causal dependencies of tasks, can be expressed by adding a place in between the two tasks, i.e. between the two transitions if we model a task with one transition, and between the stop transition of the first task and the start transition of the second task if we model according to the latter paradigm. A place in between tasks can be seen as precondition of the first task and a post-condition of the second task. In this way, places express the causal dependency between tasks and transitions.

Not only places are used for task ordering. Transitions play a role as well. For example, to express that two tasks can be executed in parallel, we need a transition with one input and two output places, and similarly if a new task can only start if two or more other tasks have completed, we need a transition with one input place for each such task and one output place to the next task. These transitions are called *AND-split* and *AND-join*, respectively. They are not considered to express tasks but *synchronization events*. Figure 6 shows the constructs. The five basic task ordering principles can be expressed in this way. In fact, most of the control flow patterns [10] can be expressed. By ordering all the tasks in this way we obtain a WFN.

Cases. A *case* is a complex entity. If we consider a classical WFN net with only one token in its initial place, then that token represents the initial state of the case. After each transition the state of the case is expressed by the marking of the WFN. Note that this marking may have more than one token. Thus, in general the state of a case is expressed by multiple tokens. If a marking is reached with only a single token in the final place, then we say that the case is in its *final state*. However, it is not always the case that the final state of a case is reachable.

In case we have two or more tokens in the initial place of a classical WFN we have to deal with *batch processing*. As with one case, the initial state of the batch is a state in which all the tokens are in the initial place, and the final state of the batch is the state in which all tokens are in the final place. Any other marking reachable form the initial state is a state of the batch. An important property of batch processing is that if we start with a batch of a certain number of cases, we should reach a final state with the same number of cases.



(a) XOR (b) AND

Fig. 6. Split and join patterns

In a batch, cases can influence each other. In fact, it is not possible to identify the case to which a token belongs. An example of batch processing in which the interference of cases is irrelevant, is the production of k identical objects. (The workflow depicts the steps in the production, and each object is represented as a case.) The production process consists of the acquisition of the parts and their assembly. So, we have to order parts in multiples of k and as long as each final product has the right number of parts, it does not matter for which product (i.e., case) they were meant. We also consider situations in which we have an unbounded *stream* of incoming cases that has to be handled. If we want to avoid interference we can use colored Petri nets in which we only use the identity attribute of the token value. Then, the batch has distinguishable cases and a transition will never consume two tokens of different cases in one firing. In this way, the cases are independent of each other.

Resources. The next concept we have to model are *resources*. As seen in Section 2.1, we classify resources into durable resources and consumable resources. Durable resources are the most important resources from a modeling point of view. Consumable resources are often anonymous supplies. The only reason to model consumable resources is to analyze the number of resources needed in the execution of cases, which is a performance issue. Simulation of the process can solve this problem. For each task we identify the amount of supplies needed. We then simulate a large number of cases to determine the distribution of the occurrences of the task, which in turn can be used to determine the resource usage.

Durable resources, like human resources, are almost always identifiable and have their own rules. For example, for human resources a separation of concerns principle should often hold, like the four-eyes principle. Durable resources are often part of a secondary business process. An important requirement is that the durable resources obey a conservation law: after handling a case, the durable resources should be available for a new case. With RCWFN (see Definition 3) we are able to express these kind of properties.

Business Rules. Last, we consider the expression of *business rules*. In practice, the context of a BP defines the boundaries within which the BP should be executed. These boundaries are defined by business rules. Often, it is difficult to verify whether a given WFN satisfies these rules. Business rules are often expressed in some form of logic, like *linear time logic* (LTL, see [71]) or *computation tree logic* (CTL or CTL* see [22]). These logics lack expressing calculations. Therefore, special languages exist to express calculation as well, like *Presburger logic* or the Business Rules Language (BRL) [36].

For verification of business rules, two methods exist: (1) verification on the level of the process model, i.e. the WFN and (2) verification on the level of a *process log*, i.e. a set of case histories. The first is called *conformance checking*. In the first situation we verify that all possible cases satisfy the business rule. To check this, the reachability graph of the Petri net is constructed and *model*

checking is applied. In many situations it is possible to use methods that exploit the Petri nets structure to speed up the verification process, like in LoLA [76]. This applies for business rules that can be expressed in LTL or CTL.

The latter method is called *auditing*. In auditing, it is easier to check complex rules, as the rule should only hold for the finite set of executed cases [7, 9, 20, 36]. However, the claim is weaker than in conformance checking: we only show absense of violations in the past. As evaluation is done on a finite set of case histories, rules expressed in Presburger logic or BRL can be evaluated.

3.3 Soundness of Workflow Nets

We often require BPs or WFNs to satisfy a set of business rules, depending on the particular context. There is one rule (with some variants) that business processes should always satisfy, independent of the context. This property is called *soundness*. Soundness is a general purpose *sanity check* for workflows. Intuitively it says that once a workflow has started it should always be able to stop without leaving "garbage" in the net. It is obvious that all processes should have this property.

There are many forms of soundness [6]. The main characteristic of most soundness concepts is *weak termination* (see Appendix A) which states that from every reachable state (marking) it is possible to reach a final state. The notion of *classical soundness*, which is defined for classical WFNs, states three important properties: (1) the workflow system should be *weakly terminating*, (2) the workflow system should have the *proper completion* property: i.e., if we reach a marking that contains the final marking, it is equal to the final marking and (3) all transitions in the model should contribute to the business goals, i.e., each transition is occurring in at least one case. The last property can also be expressed as *quasi-liveness* of the WFN. Quasi liveness is not for all forms of soundness required, although it is obvious that in practice it is not useful to have transitions in a process model that are never used.

For a classical WFN the second requirement is implied by the structure of the WFN (cf [42]), although in the first definition of soundness it was required (see [1]). The first requirement of soundness is the most essential [6]. Weak termination is actually the same problem as the *home marking property*: a marking is a home marking if and only if it is reachable form every reachable marking, which is decidable for classical Petri nets [28]. In fact, the notion of classical soundness coincides with requiring the closure to be live and bounded. A (transition-bordered) WFN-t or a (place-bordered) WFN-s is classical sound if its completion is classical sound.

Theorem 4 (Classical soundness equals liveness and boundedness [1]). *Let $N = (P, T, F, \{i\}, \{f\})$ be a classical WFN. It is classical sound if and only if the system $(\overline{N}, [i], \{[f]\})$ is live and bounded.*

As this property is the oldest result in this field and the proof gives good insights, we give a sketch of the proof.

Proof. Suppose the closure $\overline{N}$ of a WFN N is live and bounded, but not classical sound. Then the closing transition is live. Hence, a marking m and firing sequence σ exist such that $(\overline{N} : [i] \xrightarrow{\sigma} m)$ with $m > [f]$, i.e., $m = m' + [f]$ with $m' > \emptyset$. Hence, the closing transition is enabled. Firing it results in marking $m' + [i]$. Now we can repeat σ to obtain marking $m' + k.[i]$ for any k which contradicts the boundedness.

Now suppose a WFN N is classical sound but its closure $\overline{N}$ is not live nor bounded. Classical soundness implies that $[f]$ is the only marking reachable in N from $[i]$ with a token in f. Since the closing transition is only moving a token from f to i, the set of reachable markings of N is the same as of $\overline{N}$. Quasi liveness is already required by definition, so the WFN should not be bounded. This means that we have an infinite set of reachable markings. By Dickson's lemma there is an infinite sequence of reachable markings $m_1 \leq m_2 \leq ....$ Thus, $m_2 = m_1 + m'$ where $m' > 0$. This would require the existence of some firing sequence σ such that $(N : m_1 \xrightarrow{\sigma} [f])$. Hence, also $(N : m_2 \xrightarrow{\sigma} [f] + m')$ which is a contradiction. □

The closure of a WFN is interesting in itself for modeling purposes, as it models a repeatable process. Therefore we also call the closure of a WFN (classical) sound if it is live and bounded. Classical soundness can be verified using inspection of the reachability graph. We first have to determine if the reachability graph is finite, which is easy to verify: either we do not find any new marking or we encounter a marking that is containing an already found marking. In the latter case the net is unbounded and due to Theorem 4, it cannot be sound. If the reachability graph is finite, then we have to find for each state in the graph a path to the final marking, which can be done in a clever way (cf [76, 87]). The check for quasi liveness of the transitions can be performed in the same steps. In the next section, construction methods are presented that guarantee classical soundness.

A second important soundness notion is *k-soundness*. A WFN N is k-sound if in the workflow system started with k tokens in the initial place, it is always possible to reach a marking with only k tokens in the final place, i.e., the system $(N, [i^k], \{[f^k]\}$ should be weakly terminating. Note that quasi liveness is not required for k-soundness, and proper completion is implied in the same way as for classical soundness. If a WFN is 1-sound, we say the WFN is *weakly sound*. Remark that k-soundness does not imply $k + 1$-soundness nor $k - 1$-soundness, as the examples of Figures 7 and 8 show.

A WFN net is *generalized sound* if it is k-sound for all $k \in \mathbb{N}$. The verification of generalized soundness is much more difficult, as it needs a check for an infinite number of k-values. In [42], it has been shown to be decidable together with an algorithm for WFNs without inhibitor or reset arcs.

A transition-bordered or place-bordered WFN is k-sound (generalized sound) if its completion is k-sound (generalized sound). Generalized soundness is in particular important for stepwise refinement as construction method. While k-soundness is important for the processing of batches of a given size, generalized soundness is important for handling infinite streams of cases. Generalized

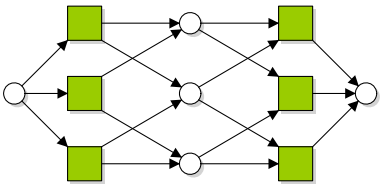


Fig. 7. 1-sound WFN, but not 2-sound

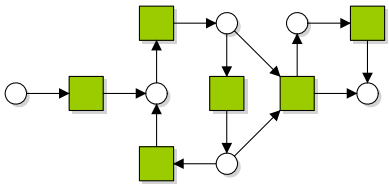


Fig. 8. 2-sound WFN, but not 1-sound

soundness states: all cases entered in the system will be handled, i.e., "what comes in will go out".

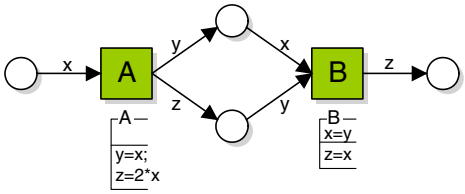
For the special classes of S-WFN and T-WFN it is proved that they are generalized sound (see [41]) by their structure. Also for FC-WFN there is a structural property: if a FC-WFN is 1-sound, then it is generalized sound (cf [46]). In the next section we will encounter more classes of WFNs that are generalized sound. We summarize these results:

Theorem 5 (Additional soundness properties). *Let* $N = (P, T, F, \{i\}, \{f\})$ *be a classical WFN.*

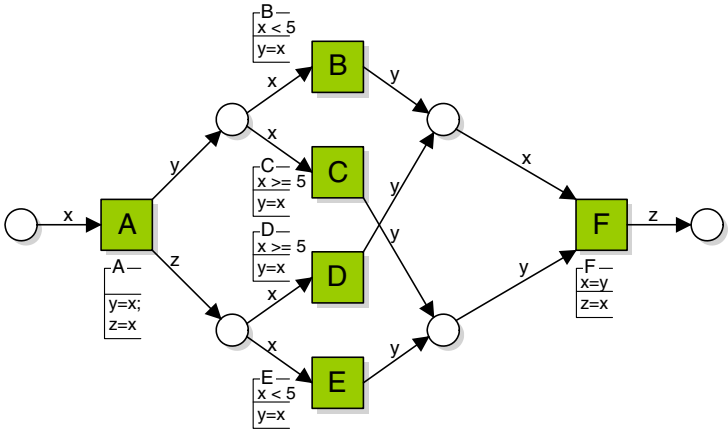
- *Generalized soundness is decidable*
- *S-WFN are generalized sound*
- *T-WFN are generalized sound*
- *If a FC-WFN is* 1*-sound, then it is generalized sound*

In literature, many different variations of soundness exist, like *up-to-k-sound*, *easy sound*, *relaxed sound* and *lazy sound*. We will discuss them briefly, for details see [6]. A WFN is up-to-k-sound if and only if it is l-sound for all $0 < l \leq k$. A WFN is easy sound if and only if it is possible that one case in isolation reaches the final marking. It is easy to verify (see [85]) that if a WFN is k-sound and easy sound, then it is up-to-k-sound. If we let a first case be handled in isolation, which is allowed by the easy soundness, then the other $k-1$ should be handled properly since the WFN is k-sound. A WFN is relaxed sound if and only if for each transition t there is a marking m reachable from an initial marking with one token in the initial place, in which t is enabled and if t fires the final marking should be reachable. Note that in a relaxed sound WFN transitions cannot be dead. There are algorithms to transform relaxed sound WFN into classical sound WFN (see [25]). Lazy soundness, considers the one case situation and requires that it must be always possible to reach a marking in which the final place has exactly one token, but there may be more tokens left (i.e., a relaxation of the proper completion property).

Up to now, we considered WFN with classical Petri net semantics, but we can also consider them with colored Petri net semantics. This gives all kind of interesting anomalies: since WFN without coloring can be sound while the colored version is unsound and vice versa, as is shown by the examples in Figure 9.



(a) Adding data makes the WFN unsound



(b) Adding data makes the WFN sound

Fig. 9. Soundness and data

Although the classical WFN of Figure 9(a) is sound, adding data makes the net unsound, as no reachable marking exist such that B is enabled in that marking. On the other hand, some behavior of the classical WFN is excluded by the data, as shown in Figure 9(b).

Similar anomalies can be created when considering time: a WFN with classical net semantics that is sound can be not sound in a timed semantics and vice versa. This means that we should be careful when we want to generalize soundness results for classical Petri nets to colored versions see [77]. When we consider only case identities then it is easier, because of the firing condition that only tokens with the same identity can be consumed and that the produced case tokens have the same identity as the consumed ones. This restricts the behavior, and in fact if a WFN is 1-sound in the classical sense then it is generalized sound if we consider case identities, since these identities make the cases independent of each other. The assumption of identifiable cases is quite natural in BP.

The last notion of soundness that we consider, is soundness for RCWFN: *resource constrained soundness* or *rc-soundness*. The intuition of soundness for RCWFNs is that the cases are handled as usual, which means that it is always

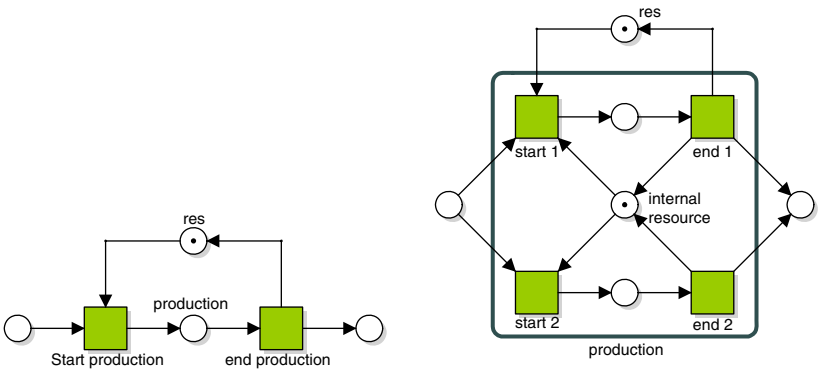
possible to reach the final marking (the marking with exactly k case tokens in the final place), if started with k cases. As we use the identity attribute, we can reformulate this requirement: the production workflow net should be 1-sound. Additionally there are (durable) resources needed to perform certain tasks, which means that transitions may consume and produce tokens for resource places. There are three important requirements: (1) when the production net reaches the final state, the resource places should have exactly the same amount of resources as at the beginning, (2) the total number of resources of any kind should not increase during the execution, and (3) if the system works properly in this sense that for a certain amount of resources per kind, then it should also work properly if we increase the number of resources of one or more kinds. Since k is an arbitrary number, we require these properties for all $k \in \mathbb{N}$. We give a formal definition:

Definition 6 (Resource constrained soundness). *Let $N = (P, R, T, F_c, F_r, \{i\}, \{f\})$ be a RCWFN with initial marking $[i^k] + r$ where $k \in \mathbb{N}$ and $r \in \mathbb{N}^R$ is a marking of the resource places only.*

- *N is (k,r)-rc-sound if for all m reachable from $[i^k] + r$, it holds that $m_R \leq r$ and marking $[f^k] + r$ is reachable.*
- *N is k-rc-sound if there is an $r \in \mathbb{N}^R$ such that N is (k, r')-rc-sound for all $r' \geq r$.*
- *N is rc-sound if it is k-rc-sound for all $k \in \mathbb{N}$.*

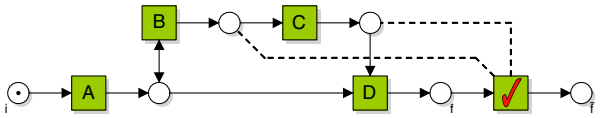
Note that if a RCWFN is sound then its production WFN is generalized sound (see [40]). For RCWFN in this form not many results are known. However for the variant with case identities, rc-soundness is decidable and there is an algorithm to verify it. Note that we only put identities to the case tokens, which make the cases independent of each other. If a RCWFN with case identities is sound then the production WFN is 1-sound. Note that the most simple form of resource modeling is when we model a task with start and stop event like in Figure 10(a) where there are one or more resource tokens dedicated for this task. It is easy to verify that any WFN where we model tasks with only one transition is branching bisimilar with the net where this transition is replaced by one of the constructs of Figure 10 when the end transitions receive the name of the task and the start transitions are silent. In this way, we can rely on the verification of (generalized) soundness of the nets without resources and one transition per task to verify rc-soundness of an RCWFN with this structure.

Up to now we did not consider inhibitor arcs or rest arcs. The verification of soundness properties when a WFN has inhibitor arcs or reset arcs is much more difficult, many questions are not decidable (see [6]). However, in the case when a WFN can always reach markings that are larger than the final marking in which only the final place is marked, the net can be made sound in a "brute force" way by adding a transition labeled ✓ that empties all places except the final place, as illustrated in Figure 11(a). Of course this is not the best way of making models!

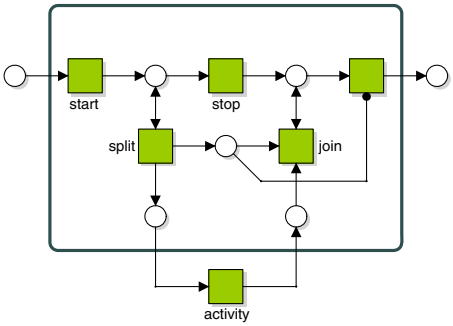


(a) Durable resource (b) Internal resource

Fig. 10. Activity with resources



(a) Reset arcs



(b) Inhibitor arcs

Fig. 11. Inhibitor arcs (bullet headed arc) and reset arcs (dashed line) repair soundness

Inhibitor arcs can also play a useful role. First note that inhibitor arcs only restrict the behavior of a system. They can destroy soundness of classical Petri nets as well as make unsound classical Petri nets sound. Consider a WFN N with inhibitor arcs and let the WFN N' be obtained from N by deleting the inhibitor arcs. If N' is bounded then N is also bounded and then we are able, by analysis of the reachability graph, to verify soundness of N. On the other hand, as inhibitor arcs only restrict behavior, it is also possible that unbounded nets become sound,

as illustrated in the example of Figure 11(b). In fact, this workflow models an important pattern to split a running case in separate instances, which in the end are combined again.

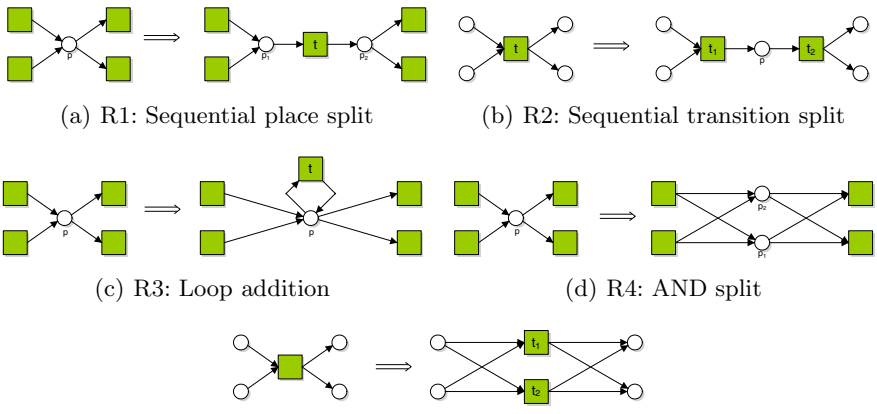
3.4 Construction Methods

Verification is an a posteriori approach: given a model, it checks whether properties like soundness hold. Although for classical Petri nets these properties are decidable, it is a very time consuming task. Therefore, we present in this section a construction method that preserves soundness: applying a rule from this method on a sound net results again in a sound net. This way, correctness of the model is guaranteed, provided that the initial model was sound.

We start with a class of WFNs that are generalized sound by their structure. First, note that a WFN consisting of a single place is generalized sound. Based on this observation, we can build a class of well-handled nets, called Jackson nets [36]. These nets are constructed with five refinement rules, as depicted in Figure 12.

The first rule (R1) is *sequential transition split*, which is shown in Figure 12(a). Given a place p in a WFN, we can split it into two new places p_1 and p_2 such that all the input transitions of p become input transitions of p_1, and, likewise, all output transitions of p become output transitions of p_2. We then add transition t such that place p_1 is the input place of t, and place p_2 is the output place of t. It is easy to verify that the refined net is generalized sound again.

The second rule (R2) is the dual of the first rule: instead of expanding a place, a transition is refined. Given a transition t in a WFN, we can split it into two new transitions t_1 and t_2 with a place in between such that the input places of t become the input places of transition t_1, place p becomes the only output place of t_1 and the only input place of transition t_2, and all output places of t in the



(a) R1: Sequential place split

(b) R2: Sequential transition split

(c) R3: Loop addition

(d) R4: AND split

(e) R5: OR split

Fig. 12. Refinement rules for Jackson nets

original net become the output places of transition t_2. The rule is depicted in Figure 12(b). We call this rule the *sequential transition split*.

To add loops in a WFN, the third rule (R3) allows to connect a new transition to an existing place in the WFN, such that this place is both the only input place and only output place of the newly added transition (see Figure 12(c)). Again, it is simple to prove that this refinement rule preserves generalized soundness. The fourth and fifth rule duplicate nodes in the WFN such that we can express AND and OR splits, respectively. The fourth rule duplicates a place (Figure 12(d)), which allows for parallelism, the fifth rule duplicates a transition (Figure 12(e)), which allows for choice.

Definition 7 (Jackson net). *A WFN is called a* Jackson net *iff it can be constructed from the singleton WFN* $(\{i\}, \emptyset, \emptyset, \{i\}, \{i\})$ *by applying the refinement rules R1,...,R5.*

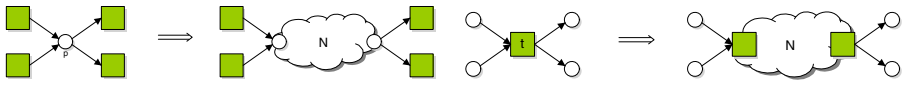
Theorem 8 (Jackson nets are generalized sound [36]). *Let N be a Jackson net. Then it is generalized sound.*

These rules are closely related to the rules of Murata [65]. In fact, when only considering refinement on nodes with a single input node and a single output node, these rules coincide. Only the rule that adds a place loop is omitted, as this rule requires the initial marking to be changed.

The nets in the class of Jackson nets are all well-formed [2], i.e., every AND split is complemented with an AND joint, and similarly for OR splits and joins. Often, this class of nets is too restrictive for modeling BP. We therefore introduce another class of workflow nets that are guaranteed to be generalized sound.

As any generalized sound net is bounded, there is an upper bound on the number of tokens in any marking reachable from the initial marking. When we refine a place by a WFN, then it should be sound for any number of tokens on the initial place of the net. Hence, if the WFN is generalized sound, we can safely refine the place with the WFN such that the refined net is generalized sound again (WP refinement, Figure 13(a)), as proven in [41]. A similar argument holds for the refinement of a transition by a WFN-t with a single initial node and a single final node (WT refinement Figure 13(b)).

Two important subclasses of WFN have been proven to be generalized sound by their structure: S-WFN and acyclic T-WFN-t with a single initial node and a single final place [41]. With these results we build a new subclass that is generalized sound by construction, which we call ST-nets. Both S-WFN and T-WFN-t are subclasses of the ST-nets. Given an ST-net, any place may be refined



(a) WP: refinement of a place by a WFN-s (b) WT: refinement of a transition by a WFN-t

Fig. 13. Place and transition refinement by a generalized WFN

by a place-bordered ST-net, and every transition may be refined by a transition-bordered ST-net. Both refinements preserve generalized soundness. Hence, any ST-net is generalized sound [41].

Definition 9 (ST-nets). *The class of* ST-nets $\mathcal{ST}$ *is recursively defined by:*

- *if* N *is an S-WFN, then* $N \in \mathcal{ST}$*;*
- *if* N *is an acyclic T-WFN-t with a single initial and final node, then* $N \in \mathcal{ST}$*;*
- *if* $N, W \in \mathcal{ST}$ *such that* W *is a WFN, and let* $p \in P_N$ *be a place of* N*. Then* $N \odot_p W \in \mathcal{ST}$*;*
- *if* $N, W \in \mathcal{ST}$ *such that* W *is a WFN-t, and let* $t \in P_N$ *be a transition of* N*. Then* $N \odot_t W \in \mathcal{ST}$*;*

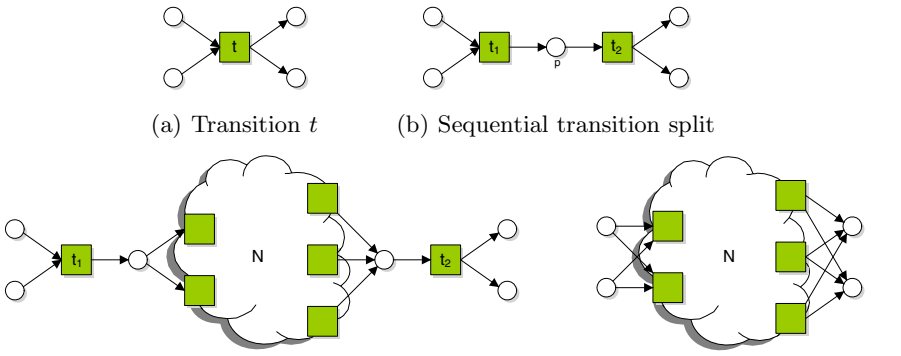
where $N \odot_n W$ *denotes the refinement of node* n *by* W*.*

Theorem 10. *Let* N *be an ST-net. Then it is generalized sound.*

It is often the case that an activity can be performed in different ways, e.g. by using different resources. To do so, we model the activity as a transition-bordered workflow net. Each initial transition represents a possible execution of the activity. We then refine the transition representing this activity by the transition-bordered workflow net. If both the original net and the refining net are generalized sound, the refined net is sound as well, which can be proven in a simple but elegant way.

Theorem 11. *Let* N *be a* k*-sound bordered WFN for some* $k \in \mathbb{N}$*. Let* W *be a generalized sound WFN-t. Then the refined net* $N \odot_t W$ *in which transition* $t \in T_N$ *is refined by* W *is* k*-sound.*

Proof. First, we refine transition t (Figure 14(a)) using the sequential place split such that we get two transitions t_1 and t_2, and a place p in between (Figure 14(b)). The resulting net is k-sound. We now refine this new place by



(a) Transition t (b) Sequential transition split

(c) Refinement with W^+ (d) Place substitution of [19]

Fig. 14. Steps in the proof of Theorem 11

the net W^+ (Figure 14(c)). By Theorem 9 of [41], the resulting net is k-sound. Places i and f of W^+ can be reduced using the "place substitution" rule of [19], which results in the net $N \odot_t W$ (Figure 14(d)). As the reduction rule preserves k-soundness, net $N \odot_t W$ is k-sound. □

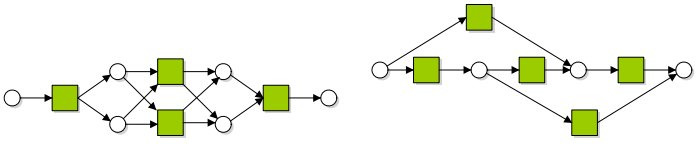
Note that the class of Jackson nets is not included in the class of ST-nets, and vice versa, as shown in Figure 15. These examples show that Jackson nets exist that are not an ST-net, and likewise, ST-nets exist that are not a Jackson net. However, the two rules, R1 and R2 are special cases of the rules WP and WT, respectively. This way, we are able to construct a new class of nets that are generalized sound by their structure.

Definition 12 (ST$^+$-Nets). *The class of* ST$^+$-Nets $\mathcal{ST}^+$ *is recursively defined by:*

- *if N is an S-WFN, then $N \in \mathcal{ST}^+$;*
- *if N is an acyclic T-WFN-t with a single initial node and a single final node, then $N \in \mathcal{ST}^+$;*
- *if $N, W \in \mathcal{ST}^+$ such that W is a WFN, and $p \in P_N$ a place of N. Then $N \odot_p W \in \mathcal{ST}^+$;*
- *if $N, W \in \mathcal{ST}^+$ such that W is a WFN-t, and $t \in P_N$ a transition of N. Then $N \odot_t W \in \mathcal{ST}^+$;*
- *if $N \in \mathcal{ST}^+$ and M can be constructed using rules R3,...,R5, then $M \in \mathcal{ST}^+$.*

Theorem 13. *Let N be an ST$^+$-net. Then it is generalized sound.*

Not all constructs needed in modeling BP can be expressed using an ST-net or a Jackson net. To guide the modeler, many patterns for modeling BP have been identified. In [10] many workflow patterns are collected based on best practices. Most of the patterns concern the control flow. We have seen already some of them. A special construct is needed for handling the multiple instance patterns, which allow for choosing the number of instances executed at runtime. One way is to model these patterns by using the splitter pattern as shown in Figure 11(b). This pattern allows to execute the activity multiple times, without specifying an upper bound at design time. It is easy to see that although the places between split and join are unbounded, the pattern is weakly terminating, i.e., placing a token in the initial place of the pattern always leads to only a single token in the final place, while all other places are empty.



(a) Jackson net, but not ST (b) ST net, but not Jackson

Fig. 15. Not all Jackson nets are ST-nets and vice versa

4 Modeling Cooperating Business Processes

Most organizations are divided into more or less autonomous cooperating business units. Each business unit can be seen as an organization. These business units cooperate to achieve the common goals of the larger organization. To realize this, the business units need to communicate. They need information from other business units to reach their goals, or on demand they can deliver requested information so that other units can accomplish their goals. As a consequence, this communication also needs to be reflected in the BPs of the different cooperating business units.

A second trend in the last decade is that organizations focus more and more on their core activities, while outsourcing all other activities. As a consequence, organizations form partnerships to achieve common goals. These organizations together form a larger organization. Again, communication is a key factor in realizing the common business goals of the larger organization. In the mean time, each organization still has its own business goals, that need to be achieved as well.

4.1 Service Oriented Approaches

Business units within an organization, or organizations in a cooperation can be seen as components of a larger system. A *component* offers services via its interfaces, and in order to deliver its services, it needs services of other components. This way, a component has two roles: a *service provider* and a *service consumer*. From a business oriented view, a component *sells* services, and in order to meet its commitments, it *buys* services of other components. In this way, a tree of components is built up to deliver a service.

Software systems have a similar component-based structure. Many different definitions exist for components. We adopt the definition of [84], which defines a *component* as a unit of composition with contractually specified interfaces and explicit context dependencies only. This definition shows two important aspects of a component: first, a component is a subsystem implementing a set of *coherent* functionalities. Its interfaces to the outside are listed explicitly in a contract, i.e., it defines how other components can access its functionality via its interfaces. Secondly, a component should be as independent as possible. A component may need other components, but these dependencies are known in advance, and they are only accessed via the specified interfaces. Note that the dependencies are on a service level, rather than defining which specific components are needed.

Current paradigms like service oriented computing (SOC) and service oriented architectures (SOA) [3,14,68] enable this business oriented view in system construction. The main principle behind SOC is to aggregate services to service compositions to implement a BP. SOA is a technology to design and execute services according to the SOC paradigm. In SOA, the network of communicating services is built at runtime, and no service knows the whole network at any point in time. An important concept in SOA is the *service broker*: a trusted third party where all components publish the services they provide (step 1 in Figure 16).

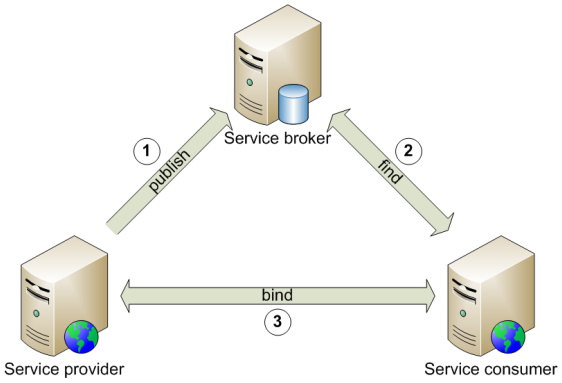


Fig. 16. SOA principle: a provider publishes it services at a broker (1). A consumer uses the broker to find a service (2) and binds to it (3).

A component that needs a service, queries the broker to find a service. The broker returns a list of possible components that deliver the requested service (step 2), after which the consumer binds to one of these components (step 3) and start communicating.

Many different formalisms and techniques exist in industry to support the modeling of service oriented paradigms. Current techniques are almost all web focused. On a low level, the Simple Object Access Protocol (SOAP) [63] defines the structure of the messages sent between components. To describe the interface of a component, the Web Service Description Language (WSDL) [21] is used. It defines the different types of messages the service can send and receive.

Each component has its own internal process to *orchestrate* its services: it defines the order in which messages are sent and received by the component. One of the main languages to model the service orchestration in a component is the Web Service Business Process Execution Language (WS-BPEL) [15]. Modeling techniques for component interaction should support the dynamics of the network formed by the components, called a *choreography*. One of the main industry standards to model this is the Web Service Choreography Language (WS-CDL) [52]. In WS-CDL, one models the possible interaction patterns between so called parties. An organization implements a party, such that it exactly mimics the behavior defined in the interaction patterns. In this article, organizations can be seen as software components and vice versa.

4.2 Modeling Interaction With Petri Nets

Communication between organizations is message driven: an organization requests a service provided by another organization, and eventually, this organization delivers it. Petri nets are well suited to model message driven communication: places in the Petri net serve as message buffers, from which messages are read in *random order*. An organization can deliver multiple services, each service can be seen as an *interface* to the outside. An interface consists of places

that are either input places for the service, i.e., *requesting messages*, or output places for the service: i.e., *sending messages*. We do not allow for places in the interface that are both input and output, as such a place can be interpreted as a shared variable, which is not allowed in asynchronous communication. We call such a Petri net with interfaces an *open net* [58, 60].

In modeling cooperating business processes, places resemble status, activity, and message buffers. The business processes of each organization (or business unit) are modeled as a WFN with interface places. As business processes are repeatable, we model a component by the fused closure of the WFN. The fused initial place and final place is called the *idle place*. If a component needs a service of another component, the corresponding interfaces are connected. As a consequence, to compose two components to model their interaction, they should only share some interfaces, and these interfaces should be their complements. In the composition, the interface places are fused and become internal places of the newly created component. The interfaces of the components that are not connected become the interfaces of the newly composed component.

As an example, consider the open net N in Figure 17. This net has three interfaces: G, H and J. Interface G consists of three output places, a,c and d, and two input places: b and e. Net M has a similar port G, with two output places, b and e, and three input places, a, c and d. Interface G of M is the complement of interface G of N. Hence, the two nets are composable with respect to interface G. In the composition, the interface places of the two components are fused based on their name and become internal places of the new component. For example, place a of N is fused with place a of M, i.e., in the composition with respect to interface G, denoted by $N \oplus_G M$, place a has the input transitions of N

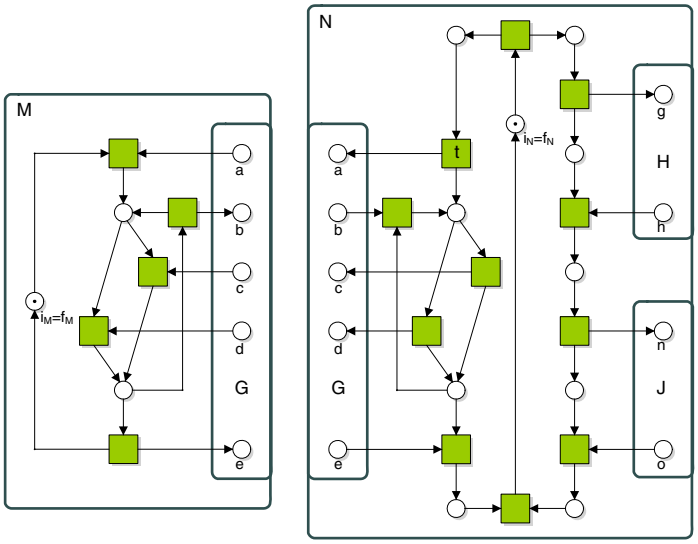


Fig. 17. Two components that share an interface

and output transitions of M. The newly created component has two interfaces, namely those of N: interface H and J. In practice, it is useful to have a rename operator in order to give places that should be fused the same name.

Many similar notions for modeling component based systems using Petri nets exist [34,54,55]. Also the Petri Net Markup Language (PNML) [47] has a module concept with interfaces. In [24], the authors propose to model choreography using Interaction Petri nets, which is a special class of Petri nets, where transitions are labeled with the source and target component, and the message type being sent. For each of the components, a Petri net with an interface is extracted. The interaction Petri net is then realizable if the composition of the behavioral interfaces is branching bisimilar related with the interaction Petri net.

4.3 Verification Methods

An important behavioral property for components is that its internal behavior should be correct: disregarding the interface, the component should be weakly terminating. Verification of asynchronously communicating components is known to be a hard problem. Because of the high degree of concurrency in such systems, model checking a complete system often becomes infeasible. A second problem is that the complete system is usually not known. Only at runtime components decide to cooperate. Therefore, *compositional verification* is needed to check these systems on conformance: given that each component is correct, and each pair of connected components satisfy some conditions, we want to conclude that the whole system is correct.

Just requiring each composition of connected pairs of components to be sound is not sufficient to guarantee the correctness of the composition, as shown in Figure 18. In this example, both $A \oplus B$ and $B \oplus C$ are sound. However the composition $A \oplus B \oplus C$ has a deadlock, since this composition introduces a cyclic dependency over the three components: after firing transition v, the system is in deadlock.

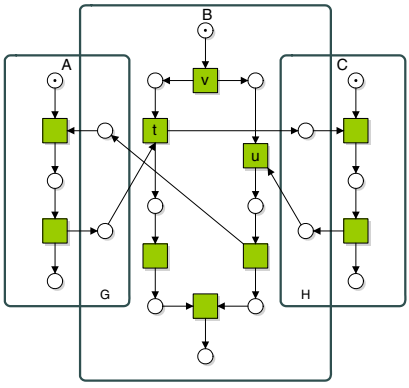


Fig. 18. The composition of three components

Behaviorally, communicating components can be seen as *partners* that together try to reach some desired state. A component cannot communicate with every partner that has the same interface. There are additional requirements on the environment based on the internal process of a component. The *operating guideline* [58] of a component is an automaton that represents all partners in a finite manner. Wolf [90] shows that if a service has a partner, a most-permissive partner exists, i.e., a partner exists such that all partners simulate this partner.

Given a service R, to check whether some service P is a partner, it should "follow" the operating guideline, i.e., partner P should simulate the operating guideline of R. However, this condition is not sufficient [90]. Therefore, each state in the operating guideline is annotated with a boolean expression stating which messages should be sent and received in each state. However, the operating guideline is currently only defined for deadlock-freedom. In [81] the authors present an extension to test whether all transitions are covered.

Within a system, if a component S is *replaced* by some other component T, the system should still function, i.e., the system should not notice the replacement. Formally, this means that every partner of S should also be a partner of T. We call this relation the *accordance pre-order* [64, 80]: T accords with S if every partner of S is a partner of T. The operating guideline can be used to decide whether a component accords to another component. In this way, the accordance pre-order can be used to decide substitutability of services.

In the setting where the network of communicating components is restricted to acyclic graphs, i.e., each component only communicate with a "parent" component, the parent component "buys" services from a child components. The accordance relation for livelock-freedom is sufficient to compositionally verify the network. However, a solution to decide accordance for weak termination is not available [79]. In [61], the authors prove that for general open nets the question whether a component has a partner is undecidable. Current research results (cf. [59, 80, 90]) are based on a message bound.

In [8, 88], the authors search for a sufficient condition to conclude weak termination based on an extra condition on the communication between each pair of communicating components. They identify several *communication patterns* that are sufficient for compositional verification of soundness. The basic communication pattern defined is the *identical communication pattern*. Given a composition of three components A, B and C, such that the composition of A and B and of B and C is sound, and A and C are disjoint, then C cannot hamper B, i.e. the composition $B \oplus C$ should mimic every firing sequence B can execute. Thus, for every firing sequence σ in B leading to its final marking, a firing sequence $\tilde{\sigma}$ should exist in the composition $B \oplus C$ that leads to the final marking of the composition, and the projection of $\tilde{\sigma}$ on the transitions of component B should be equal to σ. This way, component A does not notice whether it is communicating to component B or to the composition $B \oplus C$. The notion is closely related to simulation.

4.4 Construction Methods

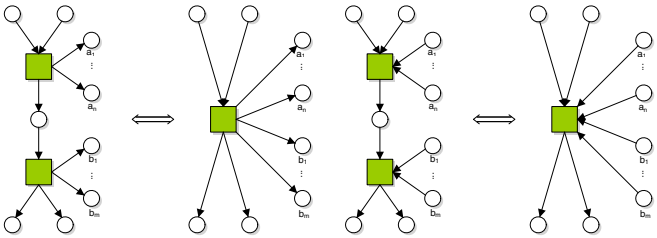
Although compositional verification of asynchronously communicating components is possible, it still is a very time consuming activity. For SOA-based architectures, i.e., in which it is unknown which components the component to be designed will cooperate with, verification needs to be done at runtime by the broker. In SOC based architectures, i.e., the components with which the component to be designed is known beforehand, construction methods exist that guarantee correctness criteria like (generalized) soundness. In this section we discuss two approaches to guarantee soundness. These construction methods allow for the construction of general networks of communicating components.

Reordering Communication. The first approach is based on public and private views of a model [11]. One first models the whole BP, and verifies soundness of the constructed WFN. Then, the WFN is divided in public views for each of the parties involved in the BP. This public view serves as a contract between the different parties. Each party implements its public view as a private view. If the private view accords to the public view, i.e., each partner of the public view is a partner of the private view, then the system composed of all the private views will be sound as well.

Like for WFN, rules exist for the refinement from a public view to a private view. The refinement rules defined in [36] for Jackson nets and the refinement rules WP and WT can be used as long as no communicating transitions, i.e., transitions that are connected to an interface, are involved. For the communicating transitions, [11] provides reordering rules that preserve accordance (see Figure 19), enabling the specialization of a public view to a private view. An overview of patterns can be found in [12].

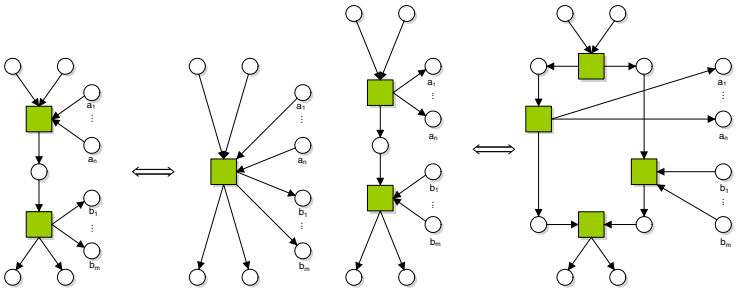
The first two rules show that a sequence of only sending transition (Figure 19(a)) or only receiving transitions (Figure 19(b)) may be refined in any order, as such a sequence can be mapped on a single transition that sends (receives) all messages at once. The third reordering rule (Figure 19(c)) shows that if first a sequence of receiving transitions occur followed by a sequence of sending transitions, the whole sequence can be replaced by a single transition. The last refinement rule (Figure 19(d)) shows that a sequence of sending transitions followed by a sequence of receiving transitions can be replaced by two parallel branches one for the sending transitions and one for the receiving transitions. In [56], the authors show that these rules can be applied to WS-BPEL, slightly relaxing the relation between abstract and executable WS-BPEL processes.

Refinement Rules. The rules presented in [11] show that accordance is preserved under the reordering rules. However, these rules do not extend the behavior of the components. In the remainder of this section, we will present three refinement rules that extend the behavior of a network of communicating components [44,88]: The first rule refines a single component, the second rule refines the communication between two components, and the last rule introduces a new component in the network.



(a) Rule C1: reordering send actions

(b) Rule C2: reordering receiving actions



(c) Rule C3: Sequence of first receive then send

(d) Rule C4: Sequence of first send then receive

Fig. 19. Reordering communication

Within a business process modeled as a Petri net, a place can resemble both an activity or a state. The place refinement defined in Section 3.4 allows us to refine a place with another WFN that is generalized sound. As for open nets the concept of generalized soundness is not defined, we require the place that will be refined to be safe: if the place is safe, it may be refined by an open net whose inner structure is a WFN. The resulting net has both the interfaces of the original net as well as the interfaces of the refining net, as shown in Figure 20.

This definition of place refinement propagates the ports of the refining component to the original component. At a first glance, this definition seems to contradict the paradigm of information hiding. However, the definition allows for the refinement of a component by a composed component, as long as this composition remains a workflow component. This way, the ports remain invisible to the environment of the original component.

In the refinement of a system of asynchronously communicating components, different components can contain places that together represent a single procedure. To refine the system with the actual procedure, which mostly includes communication between the different components, these places need to be simultaneously refined by the actual procedure.

Not every subset of places in such a system can be refined while preserving soundness. First, a marking in the system should exist in which all places to be

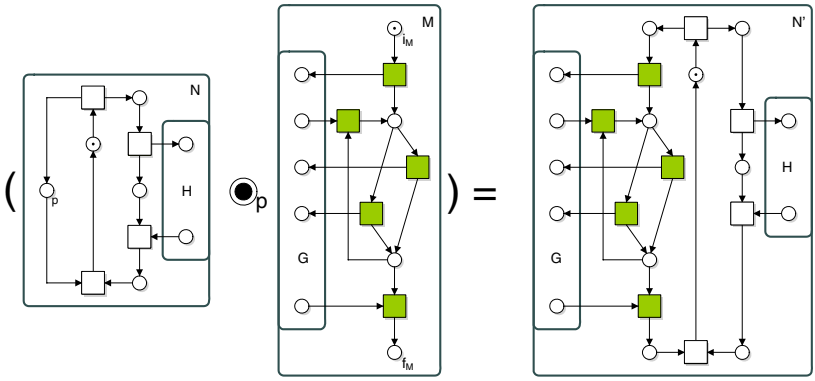


Fig. 20. Refinement of place p in N by component M

refined are marked. Second, such a place can only become marked again after all other places to be refined have been marked, i.e., when refining these places by rule R1 (Figure 12(a)) the synchronic distance [83] of the newly added transitions should be 1. Thirdly, from any marking reachable a path exists that ensures that all places to be refined are marked before one of these places become unmarked again.

Figure 21 depicts a May/Exit transition system that expresses the three conditions. Solid transitions are exit transitions, and from every state in the system it should be possible to reach the final marking using only exit transitions. Each state is annotated by two sets of places: the set of places that has been marked in the current cycle and the set of places that already have become unmarked. The transitions p and q in this system are mapped onto the transitions in the preset of the places p and q, respectively, and the transitions p' and q' are mapped onto the transitions in the preset of the places p and q, respectively. All other transitions are mapped to the silent transition. If every firing sequence in the Petri net can be replayed in the May/Exit transition system, and for each marking a firing sequence to the final marking exist using only the solid transitions of the May/Exit transition system, then the places p and q are synchronizable [38, 45].

Refinement of a set of places by an actual procedure can be seen as the refinement by a *communication protocol* between the components. A communication protocol is a set of open nets whose inner structure is a WFN, and each open net has at most one interface for each of the other open nets in the protocol, and no other interfaces exist. In this way, a communication protocol models the cooperation between the different parties.

Using the concept of synchronizable places and the communication protocol, we can define a refinement rule for the communication between components. Given a system of communicating components, a set of synchronizable places and a communication protocol between as many parties as there are synchronizable places. In the refinement, we refine each synchronizable place with a party from

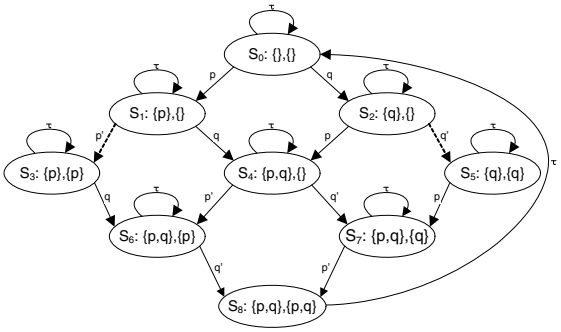


Fig. 21. May/Exit transition system for synchronizable places p and q

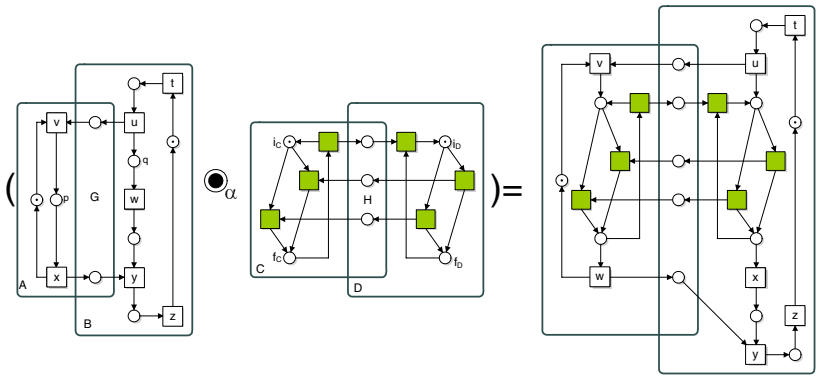


Fig. 22. Refinement of synchronizable places p and q with $\alpha = \{p \mapsto C, q \mapsto D\}$: place p is refined by C and place q simultaneously by D

the communication protocol. If the communication protocol is sound, the refined net is sound again.

Whereas the previous two rules focused on the extension of existing components, these rules do not allow for expansion of the network by adding new components. With the next rule, it is possible to connect new components in a system such that the system remains sound. The rule is based on the principle of *outsourcing*.

Consider Figure 23. For example, if place p has the meaning that when a token resides in it, "an item is produced", and the decision is taken to outsource the production activity, we can add two transitions: a "start producing item" and a "finish producing item". Then the start transition initiates the component producing the item, and the finish transition fires if the item is produced. In this way, we create a new interface for outsourcing the activity, which allows us to connect it to a new component. In fact, we replace place p by a communication protocol between two parties: the existing component and a new component.

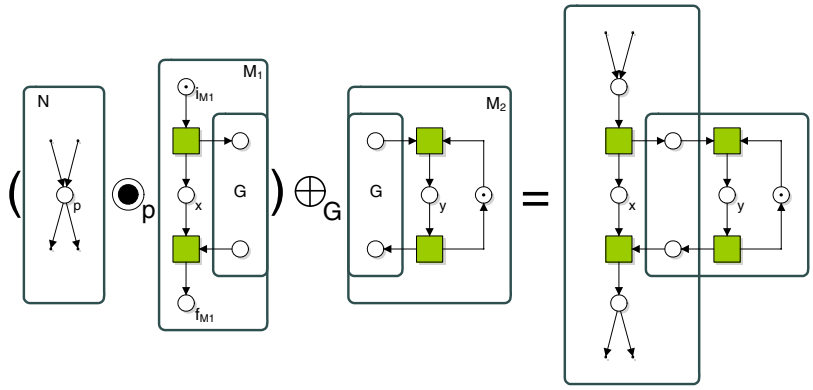


Fig. 23. Extending the composition with a new coupled component

The new component is formed by taking the fused closure of the open net in the communication protocol. If every firing sequence from the idle state of the new component back to this idle state starts with receiving a message and ends with sending a message, then the refinement preserves soundness.

4.5 Communication Protocols

The last two refinement rules presented in the previous section are based on sound communication protocols. Two special subclasses of WFNs are the class of acyclic marked graphs and the class of state machines. These nets are generalized sound by their structure [41]. In this section, we will discuss a subclass of communication protocols based on these classes of WFNs that are sound by their structure. We focus here on sound communication protocols between two parties.

Soundness of a communication protocol means that all parties should always be able to reach their final marking, and no messages are pending in the interfaces. Every communication protocol is a WFN-s net. As the completion of a WFN-s is bisimilar to that WFN-s when hiding the initial and final transitions, soundness of the WFN implies soundness of the communication protocol.

As proven in [41], acyclic marked graphs are generalized sound and safe by their structure. Now let us consider an open net whose inner structure is a T-WFN, and each interface place is connected to exactly one transition, then the composition of two of these nets is again a marked graph. Hence, if the composition is acyclic, the WFN of the communication protocol is sound, and thus also the communication protocol. As a result, the class of communication protocols whose inner structure is an acyclic marked graph are sound by their structure. We name this class AT-nets.

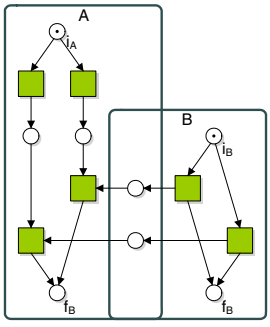
As communication protocols are also asynchronously communicating systems, we can apply the refinement rules of the previous section on them. However,

as the refinement over components requires synchronizability of the places to be refined, we still need to check which pairs of places are synchronizable. In an acyclic marked graph, every place will be marked exactly once. Hence, if a marking exists in which two places are marked at the same time, these places are synchronizable. For this subclass of communication protocols, we can express this as a graph property: two places are synchronizable if there does not exist a directed path between the two places. Hence, the set of all synchronizable places can be computed from the structure of the Petri net.

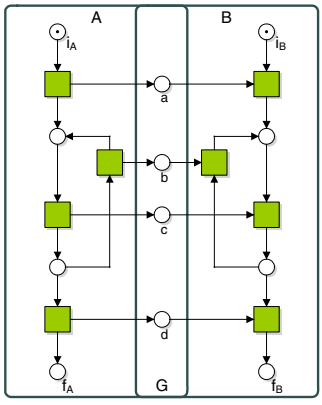
A second class of WFNs that is generalized sound by structure is the class of state machine WFNs. Consider an open net whose inner structure is a S-WFN. Then unlike the composition of open nets whose inner structure is T-WFN, the composition of two of these open nets is not a state machine. Composing two state machines introduces concurrency; it is very simple to compose two open nets with an inner S-WFN structure resulting in a composition that is not sound.

For example, consider the example of Figure 24(a). In this net, first component A makes a decision, then component B makes a decision. As both decisions are independent, if B makes the wrong choice, the composition reaches a deadlock. A solution to overcome this problem is to only connect isomorphic open nets with an inner S-WFN structure such that the interface places are determined by the isomorphism relation, and transitions in conflict either all send or receive. However, as Figure 24(b) shows, this is not sufficient. Also the direction of the communication matters: any loop should contain at least one sending and one receiving transition, otherwise the composition can become unbounded.

These observations show that compositions of open net with an inner S-WFN structure should "agree" on the isomorphism: each transition should only communicate with its isomorphic counter part, all transitions in conflict



(a) Decisions should be "in sync"



(b) Direction of communication is important

Fig. 24. unsound compositions of state machine components

either all send a message or all receive a message, and each loop contains at least one sending and one receiving transition. We call this set of communication protocols IS-nets.

Markings of such a composition have a special property: as each of the components is an S-WFN, any marking in the composition can be split into a marked place of the first component, a marked place of the second component and some interface places are marked. As each loop contains both a sending and a receiving transition, the composition is safe: no interface place can contain more than one token. Therefore, if the composition agrees on the isomorphism, it is always possible to reach a marking in which the interface places are empty, and, if this is the case, the only marked places are their isomorphic counter parts. As a result, if the composition of two isomorphic open nets with an inner S-WFN structure agrees on the isomorphism, it is both safe and sound. A direct consequence of this property is that each pair of a place and its isomorphic counterpart is also synchronizable.

Based on the class of AT-nets and IS-nets, we recursively define a larger class of sound and safe communication protocols called ATIS-nets. Using the refinement rule over components, we can refine any two synchronizable places by a sound and safe communication protocol. Hence, if we refine two synchronizable places in an ATIS net with an ATIS net, the result is sound and safe again.

5 Conclusions

We have shown how relevant aspects of business processes can be modeled with Petri nets and explained why soundness, or weak termination, is an important sanity check for process modeling. We have presented a number of model construction rules allowing to develop sound by construction business processes. Some of these rules are refinement rules, where a node of a Petri net gets refined by another Petri net, or composition rules, where the original nets become subnets of the composed net. These construction rules can be combined with workflow patterns, which in fact define best practices.

Besides single business processes we also considered the modeling of cooperating, communicating, business processes. This topic is especially important since in practice, business processes seldom operate in isolation. We extended the construction rules for single business processes to construction rules for sets of communicating business processes. Since communicating business processes can be transformed into one big business process, the construction methods for communicating business processes can be used for modeling of single business processes as well.

Business processes do not only occur in physical organizations but also in software systems. If software systems are developed according to the paradigm of service oriented computing, then components that deliver services to each other form a direct analogy with cooperating organizations. Each component has an internal orchestration process, which is in fact a business processes, while sets of communicating components are in fact communicating business processes.

The whole system can be considered as one organization and all the components as business units. So the theory developed in this paper can be applied to component-based software systems as well.

Acknowledgements. The authors would like to thank Christian Stahl for the useful discussions and his valuable comments on cooperating business processes.

A Basic Notations

Let S be a set. The powerset of S is denoted by $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$. We use $|S|$ for the number of elements in S. Two sets U and V are *disjoint* if $U \cap V = \emptyset$. We denote the cartesian product of two sets S and T by $S \times T$. On a cartesian product we define two projection functions $\pi_1 : S \times T \to S$ and $\pi_2 : S \times T \to T$ such that $\pi_1((s,t)) = s$ and $\pi_2((s,t)) = t$ for all $(s,t) \in S \times T$. We lift the projection function to sets in the standard way.

A *bag* m over S is a function $m : S \to \mathbb{N}$, where $\mathbb{N} = \{0, 1, \ldots\}$ denotes the set of natural numbers. We denote e.g. the bag m with an element a occurring once, b occurring three times and c occurring twice by $m = [a, b^3, c^2]$. The set of all bags over S is denoted by $\mathbb{N}^S$. Sets can be seen as a special kind of bag where all elements occur only once; we interpret sets in this way whenever we use them in operations on bags. We use $+$ and $-$ for the sum and difference of two bags, and $=, <, >, \leq, \geq$ for the comparison of two bags, which are defined in a standard way. The projection of a bag $m \in \mathbb{N}^S$ on elements of a set $U \subseteq S$, is denoted by $m_{|U}$, and is defined by $m_{|U}(u) = m(u)$ for all $u \in U$ and $m_{|U}(u) = 0$ for all $u \in S \setminus U$.

A *sequence* over S of length $n \in \mathbb{N}$ is a function $\sigma : \{1, \ldots, n\} \to S$. If $n > 0$ and $\sigma(i) = a_i$ for $i \in \{1, \ldots, n\}$, we write $\sigma = \langle a_1, \ldots, a_n \rangle$. The length of a sequence is denoted by $|\sigma|$. The sequence of length 0 is called the *empty sequence*, and is denoted by ϵ. The set of all finite sequences over S is denoted by S^*. Let $\nu, \gamma \in S^*$ be two sequences. *Concatenation*, denoted by $\sigma = \nu; \gamma$ is defined as $\sigma : \{1, \ldots, |\nu| + |\gamma|\} \to S$, such that for $1 \leq i \leq |\nu|$: $\sigma(i) = \nu(i)$, and for $|\nu| + 1 \leq i \leq |\nu| + |\gamma|$: $\sigma(i) = \gamma(i - |\nu|)$. *Projection* of a sequence $\sigma \in S^*$ on elements of a set $U \subseteq S$, denoted by $\sigma_{|U}$, is inductively defined by $\epsilon_{|U} = \epsilon$ and $(\langle a \rangle; \sigma)_{|U} = \langle a \rangle; \sigma_{|U}$ if $a \in U$ and $(\langle a \rangle; \sigma)_{|U} = \sigma_{|U}$ otherwise. The *Parikh vector* of a sequence σ, denoted by $\overrightarrow{\sigma}$ is inductively defined by $\overrightarrow{\epsilon} = \emptyset$ and $\overrightarrow{\langle a \rangle; \sigma} = [a] + \overrightarrow{\sigma}$ for all $a \in S$.

If we give a tuple a name, we subscript the elements with the name of the tuple, e.g. for $N = (A, B, C)$ we refer to its elements by A_N, B_N, and C_N. If the context is clear, we omit the subscript.

Labeled Transition Systems. A *labeled transition system* (LTS) is a 5-tuple $(S, \mathcal{A}, \longrightarrow, s_i, \Omega)$ where

- S is a set of *states*;
- $\mathcal{A}$ is a set of *actions*;
- $\longrightarrow \subseteq S \times (\mathcal{A} \cup \{\tau\}) \times S$ is a *transition relation*, where $\tau \notin \mathcal{A}$ is the silent action [17];
- $s_i \in S$ is the *initial state*; and
- $\Omega \subseteq S$ is the set of *final states*, also called *accepting states*.

Let $L = (S, \mathcal{A}, \longrightarrow, s_i, \Omega)$ be an LTS. For $s, s' \in S$ and $a \in \mathcal{A} \cup \{\tau\}$, we write $(L : s \xrightarrow{a} s')$ iff $(s, a, s') \in \longrightarrow$. If $(L : s \xrightarrow{a} s')$, we say that state s' is *reachable* from s by an action labeled a. A state $s \in S$ is called a *deadlock* if no action $a \in \mathcal{A} \cup \{\tau\}$ and state $s' \in S$ exist such that $(L : s \xrightarrow{a} s')$. We define $\Longrightarrow$ as the smallest relation such that $(L : s \Longrightarrow s')$ if $s = s'$ or $\exists s'' \in S : (L : s \Longrightarrow s'' \xrightarrow{\tau} s')$. As a notational convention, we may write $\xRightarrow{\tau}$ for $\Longrightarrow$. For $a \in \mathcal{A}$ we define $\xRightarrow{a}$ as the smallest relation such that $(L : s \xRightarrow{a} s')$ if $\exists s_1, s_2 \in S : (L : s \Longrightarrow s_1 \xrightarrow{a} s_2 \Longrightarrow s')$. We lift the notations of actions to sequences. For the empty sequence ϵ, we have $(L : s \xrightarrow{\epsilon} s')$ iff $(L : s \Longrightarrow s')$. A sequence $\sigma \in \mathcal{A}^*$ of length $n > 0$ is a firing sequence from $s_0, s_n \in S$, denoted by $(L : s_0 \xrightarrow{\sigma} s_n)$ if states $s_{j-1}, s_j \in S$ exist such that $(L : s_{j-1} \xRightarrow{\sigma(j)} s_j)$ for all $1 \leq j \leq n$. If a firing sequence σ exists such that $(L : s \xrightarrow{\sigma} s')$ we say that s' is *reachable* from s. The set of all reachable states from s are the states from the set $\mathcal{R}(L, s) = \{s' \mid \exists \sigma \in \mathcal{A}^* : (L : s \xrightarrow{\sigma} s')\}$.

An LTS $L = (S, \mathcal{A}, \longrightarrow, s_i, \Omega)$ is *weakly terminating* if $\Omega \cap \mathcal{R}(L, s) \neq \emptyset$ for all states $s \in \mathcal{R}(L, s_i)$, i.e. from every state reachable from the initial state some final marking can be reached.

Definition 14 (Hiding). *Let $L = (S, \mathcal{A}, \longrightarrow, s_i, \Omega)$ be an LTS. Let $H \subseteq \mathcal{A}$. We define the operation τ_H on an LTS by $\tau_H(L) = (S, \mathcal{A} \setminus H, \longrightarrow', s_i, \Omega)$, where for $m, m' \in S$ and $a \in \mathcal{A}$ we have $(m, a, m') \in \longrightarrow'$ if and only if $(m, a, m') \in \longrightarrow$ and $a \notin H$ and $(m, \tau, m') \in \longrightarrow'$ if and only if $(m, \tau, m') \in \longrightarrow$ or $(m, a, m') \in \longrightarrow$ and $a \in H$.*

An LTS L' delay simulates an LTS L if in every two related states, each action L can do, LTS L' can perform as well, possibly after some silent steps. If both L' simulates L and L simulates L' with simulation relations R and R^{-1}, we say L and L' are delay bisimilar.

Definition 15 (Delay (bi)simulation). *Let $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ and $L' = (S', \mathcal{A}', \rightarrow', s_i', \Omega')$ be two LTSs. The relation $Q \subseteq S \times S'$ is a* delay simulation, *denoted by $L \preceq_Q L'$, if:*

1. $s_i \, Q \, s_i'$;
2. $\forall s_1, s_2 \in S, a \in \mathcal{A} \cup \{\tau\}, s_1' \in S' : ((L : s_1 \xrightarrow{a} s_2) \wedge s_1 \, Q \, s_1') \implies (\exists s_2' \in S' : (L' : s_1' \xRightarrow{a} s_2') \wedge s_2 \, Q \, s_2')$; *and*
3. $\forall s' \in S', s_f \in \Omega : s_f \, Q \, s' \implies s' \in \Omega$.

If both Q and Q^{-1} are delay simulations, Q is a delay bisimulation *denoted by $L \simeq_Q L'$.*

In the remainder, if we use the term (bi)simulation, we refer to delay (bi)simulation. For a more elaborate overview of simulation relations, we refer the reader to [29].

Petri Nets. A *Petri net* N is a 3-tuple (P, T, F) where (1) P and T are two disjoint sets of *places* and *transitions* respectively; (2) $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the *flow function*. The elements from the set $P \cup T$ are called the *nodes* of N. A pair $(n_1, n_2) \in (P \times T) \cup (T \times P)$ is called an *arc* if $F(n_1, n_2) > 0$. Places are depicted as circles, transitions as squares. For each pair $(n_1, n_2) \in (P \times T) \cup (T \times P)$ such that $F(n_1, n_2) > 0$, an arc is drawn from n_1 to n_2. Two Petri nets $N = (P, T, F)$ and $N' = (P', T', F')$ are *disjoint* if and only if $(P \cup T) \cap (P' \cup T') = \emptyset$. Let $N = (P, T, F)$ be a Petri net. Given a node $n \in (P \cup T)$, we define its *preset* ${}^\bullet_N n = \{n' \mid F(n', n) > 0\}$, and its *postset* $n^\bullet_N = \{n' \mid F(n, n') > 0\}$. We lift the notation of preset and postset to sets and sequences. Given a set $U \subseteq (P \cup T)$, ${}^\bullet_N U = \bigcup_{n \in U} {}^\bullet_N n$ and $U^\bullet_N = \bigcup_{n \in U} n^\bullet_N$. The preset of a sequence $\sigma \in T^*$ is the set of all places that occur in a preset of a transition in σ, i.e., ${}^\bullet_N \sigma = \{p \mid \exists 1 \leq i \leq |\sigma| : p \in {}^\bullet_N \sigma(i)\}$. Likewise, the postset of σ is the set of all places that occur in a postset of a transition in σ, i.e., $\sigma^\bullet_N = \{p \mid \exists 1 \leq i \leq |\sigma| : p \in \sigma(i)^\bullet_N\}$. If the context is clear, we omit the N in the subscript.

Let $N = (P, T, F)$ be a Petri net. A *marking* of N is a bag $m \in \mathbb{N}^P$, where $m(p)$ denotes the number of *tokens* in place $p \in P$. If $m(p) > 0$, place $p \in P$ is called *marked* in marking m. A Petri net N with corresponding marking m is written as (N, m) and is called a *marked Petri net*. A *system* $\mathcal{N}$ is a 3-tuple $((P, T, F), m_0, \Omega)$ where $((P, T, F), m_0)$ is a marked Petri net and $\Omega \subseteq \mathbb{N}^P$ is a set of *final markings*.

The semantics of a system $\mathcal{N} = ((P, T, F), m_0, \Omega)$ is defined by an LTS $\mathcal{S}(\mathcal{N}) = (\mathbb{N}^P, T, \to, m_0, \Omega)$ where $(m, t, m') \in \longrightarrow$ iff $F(p, t) \leq m(p)$ and $m'(p) = m(p) - F(p, t) + F(t, p)$ for all $p \in P$, $m, m' \in \mathbb{N}^P$ and $t \in T$. We write $(N : m \xrightarrow{t} m')$ as a shorthand notation for $(\mathcal{S}(\mathcal{N}) : m \xrightarrow{t} m')$ and $\mathcal{R}(\mathcal{N}, m)$ for $\mathcal{R}(\mathcal{S}(\mathcal{N}), m)$.

Let $\mathcal{N} = ((P, T, F), m_0, \Omega)$ be a system. Place p is *k-bounded* in $\mathcal{N}$ for some $k \in \mathbb{N}$, if $m(p) \leq k$ for any marking $m \in \mathcal{R}(\mathcal{N}, m_0)$. If all places are k-bounded, we say that the system is k-bounded. A system is bounded if there exists a $k \in \mathbb{N}$ such that the system is k-bounded. A transition $t \in T$ is *live* in $\mathcal{N}$ if for all markings $m \in \mathcal{R}(\mathcal{N}, m_0)$ a $\sigma \in T^*$ and $m' \in \mathbb{N}^P$ exist such that $(N : m \xrightarrow{\sigma} m')$ and $(N : m' \xrightarrow{t})$. If all transitions of a system are live, the system is called live. A transition $t \in T$ is *quasi-live* in $\mathcal{N}$ if there exists a reachable marking $m \in \mathcal{R}(\mathcal{N}, m_0)$ such that $(N : m \xrightarrow{t})$. If all transitions in the system are quasi-live, the system is called quasi-live.

Weak termination of a system corresponds to weak termination of the corresponding transition system.

A Petri net $N = (P, T, F)$ is a *marked graph* if $|{}^\bullet t| \leq 1$ and $|t^\bullet| \leq 1$ for all transitions $t \in T$. It is a *state machine* if $|{}^\bullet p| \leq 1$ and $|p^\bullet| \leq 1$ for all places $p \in P$.

References

1. van der Aalst, W.M.P.: Verification of Workflow Nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
2. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
3. van der Aalst, W.M.P., Beisiegel, M., van Hee, K.M., König, D., Stahl, C.: An SOA-Based Architecture Framework. International Journal of Business Process Integration and Management 2(2), 91–101 (2007)
4. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., de Medeiros, A.K.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W(E.), Weijters, A.J.M.M.T.: ProM 4.0: Comprehensive Support for *Real* Process Analysis. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 484–494. Springer, Heidelberg (2007)
5. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods and Systems. Academic Service, Schoonhoven (1997)
6. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. In: Formal Aspects of Computing, pp. 1–31 (2010)
7. van der Aalst, W.M.P., van Hee, K.M., van der Werf, J.M.E.M., Verdonk, M.: Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor. IEEE Computer 43(3), 102–105 (2010)
8. van der Aalst, W.M.P., van Hee, K.M., Massuthe, P., Sidorova, N., van der Werf, J.M.E.M.: Compositional Service Trees. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 283–302. Springer, Heidelberg (2009)
9. van der Aalst, W.M.P., van Hee, K.M., van der Werf, J.M.E.M., Kumar, A., Verdonk, M.C.: Conceptual model for online auditing. Decision Support Systems 50(3), 636–647 (2011)
10. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Advanced workflow patterns. In: Scheuermann, P., Etzion, O. (eds.) CoopIS 2000. LNCS, vol. 1901, pp. 18–29. Springer, Heidelberg (2000)
11. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: From Public Views to Private Views – Correctness-by-Design for Services. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 139–153. Springer, Heidelberg (2008)
12. van der Aalst, W.M.P., Mooij, A.J., Stahl, C., Wolf, K.: Service Interaction: Patterns, Formalization, and Analysis. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 42–88. Springer, Heidelberg (2009)
13. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data & Knowlegde Engineering 53(2), 129–162 (2005)
14. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services – Concepts, Architectures and Applications. Springer, Heidelberg (2004)
15. Alves, A., Arkin, A., Askary, S., et al.: Web Services Business Process Execution Language Version 2.0 (OASIS Standard). WS-BPEL TC OASIS (2007), `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
16. Araki, T., Kasami, T.: Some decision problems related to the reachability problem for petri nets. Theor. Computer Science 3, 85–104 (1977)
17. Basten, T., van der Aalst, W.M.P.: Inheritance of Behavior. Journal of Logic and Algebraic Programming 47(2), 47–145 (2001)

18. Beisiegel, M., Khand, K., Karmarkar, A., Patil, S., Rowley, M.: Service Component Architecture Assembly Model Specification Version 1.1 (2010)
19. Berthelot, G.: Transformations and Decompositions of Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 360–376. Springer, Heidelberg (1987)
20. Chan, D.Y., Vasarhelyi, M.A.: Innovation and practice of continuous auditing. International Journal of Accounting Information Systems 12(2), 152–160 (2011)
21. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1 (2001), http://www.w3.org/TR/wsdl
22. Clarke, E., Emerson, E.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
23. Decker, G., Overdick, H., Weske, M.: Oryx – An Open Modeling Platform for the BPM Community. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 382–385. Springer, Heidelberg (2008)
24. Decker, G., Weske, M.: Local Enforceability in Interaction Petri Nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007)
25. Dehnert, J., Rittgen, P.: Relaxed soundness of business processes. In: Dittrich, K.R., Geppert, A., Norrie, M. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 157–170. Springer, Heidelberg (2001)
26. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press (1995)
27. Desel, J., Reisig, W., Rozenberg, G. (eds.): Lectures on Concurrency and Petri Nets. LNCS, vol. 3098. Springer, Heidelberg (2004)
28. Frutos-Escrig, D., Johnen, C.: Decidability of home space property. Technical Report 503, LRI (1989)
29. van Glabbeek, R.J.: The Linear Time - Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
30. Goud, R., van Hee, K.M., Post, R.D.J., van der Werf, J.M.E.M.: Petriweb: a Repository for Petri Nets. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 411–420. Springer, Heidelberg (2006)
31. Object Management Group. Unified Modeling Language: Superstructure, version 2.0 (August 2005)
32. Hammer, M.: Re-engineering Work: Don't automate, Obliterate. Harvard Business Review, 104–112 (July/August 1990)
33. Hammer, M., Champy, J.: Re-engineering the Corporation. Nicolas Brealy Publishing, London (1993)
34. Heckel, R.: Open Petri Nets as Semantic Model for Workflow Integration. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 281–294. Springer, Heidelberg (2003)
35. van Hee, K.M.: Information System Engineering - A formal approach. Cambridge University Press (1994)
36. van Hee, K., Hidders, J., Houben, G.-J., Paredaens, J., Thiran, P.: On-the-Fly Auditing of Business Processes. In: Jensen, K., Donatelli, S., Koutny, M. (eds.) ToPNoC IV. LNCS, vol. 6550, pp. 144–173. Springer, Heidelberg (2010)
37. van Hee, K.M., Keiren, J., Post, R., Sidorova, N., van der Werf, J.M.E.M.: Designing Case Handling Systems. In: Jensen, K., van der Aalst, W.M.P., Billington, J. (eds.) ToPNaC I. LNCS, vol. 5100, pp. 119–133. Springer, Heidelberg (2008)

38. van Hee, K.M., Mooij, A.J., Sidorova, N., van der Werf, J.M.E.M.: Soundness-Preserving Refinements of Service Compositions. In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 131–145. Springer, Heidelberg (2011)
39. van Hee, K.M., Post, R.D.J., Somers, L.J.: Yet Another Smart Process Editor. In: European Simulation and Modelling Conference 2005, pp. 527–530 (2005)
40. van Hee, K.M., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Soundness of Resource-Constrained Workflow Nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 250–267. Springer, Heidelberg (2005)
41. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003)
42. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Generalised Soundness of Workflow Nets Is Decidable. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 197–215. Springer, Heidelberg (2004)
43. van Hee, K.M., Sidorova, N., Voorhoeve, M., van der Werf, J.M.E.M.: Generation of Database Transactions with Petri nets. Fundamenta Informatica 93(1-3), 171–184 (2009)
44. van Hee, K.M., Sidorova, N., van der Werf, J.M.E.M.: Construction of Asynchronous Communicating Systems: Weak Termination Guaranteed! In: Baudry, B., Wohlstadter, E. (eds.) SC 2010. LNCS, vol. 6144, pp. 106–121. Springer, Heidelberg (2010)
45. van Hee, K.M., Sidorova, N., van der Werf, J.M.E.M.: Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved! In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 149–168. Springer, Heidelberg (2011)
46. van Hee, K.M., Voorhoeve, M.: Soundness of free choice workflow nets. In: Formal Approaches to Business Processes and Web Services - International Workshop (2007)
47. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Treves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. Petri Net Newsletter 76, 9–28 (2009)
48. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
49. ter Hofstede, A.H.M., van der Aalst, W.M.P., Adams, M., Russell, N.: Modern Business Process Automation: YAWL and its Support Environment. Springer, Berlin (2010)
50. Holzmann, G.J.: SPIN Model Checker, The: Primer and Reference Manual. Addison-Wesley Professional (2004)
51. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Berlin (2009)
52. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web Services Choreography Description Language Version 1.0 (November 2005), `http://www.w3.org/TR/ws-cdl-10/`
53. Keller, G., Nüttgens, N., Scheer, A.W.: Semantische Process- modellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89, University of Saarland, Saarbrücken (1992) (in German)
54. Kindler, E., Petrucci, L.: Towards a Standard for Modular Petri Nets: A Formalisation. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 43–62. Springer, Heidelberg (2009)
55. Kindler, E.: A Compositional Partial Order Semantics for Petri Net Components. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 235–252. Springer, Heidelberg (1997)

56. König, D., Lohmann, N., Moser, S., Stahl, C., Wolf, K.: Extending the Compatibility Notion for Abstract WS-BPEL Processes. In: 17th International Conference on World Wide Web (WWW 2008), pp. 785–794. ACM (April 2008)
57. La Rosa, M., Reijers, H.A., van der Aalst, W.M.P., Dijman, R.M., Mendling, J., Dumas, M., García-Bañuelos, L.: APROMORE: An advanced process model repository. Expert Systems with Applications 38(6), 7029–7040 (2011)
58. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)
59. Massuthe, P.: Operating Guidelines for Services. PhD thesis, Technische Universiteit Eindhoven (2009)
60. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics 1(3), 35–43 (2005)
61. Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Undecidability of partner existence for open nets. Information Processing Letters 108(6), 374–378 (2008)
62. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Berlin (1980)
63. Miltra, N., Lafon, Y.: Soap version 1.2 part 0: Primer, 2nd edn. (2007), http://www.w3.org/TR/2007/REC-soap12-part0-20070427/
64. Mooij, A.J., Parnjai, J., Stahl, C., Voorhoeve, M.: Constructing Replaceable Services Using Operating Guidelines and Maximal Controllers. In: Bravetti, M., Bultan, T. (eds.) WS-FM 2010. LNCS, vol. 6551, pp. 116–130. Springer, Heidelberg (2011)
65. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (1989)
66. Object Management Group. Business Process Modeling Notation, V1.1 (2008), http://www.omg.org/spec/BPMN/1.1/PDF/
67. Object Management Group. Semantics of Business Vocabulary and Business Rules (SBVR), v1.0 (2008), http://www.omg.org/spec/SBVR/1.0/PDF/
68. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson-Prentice Hall (2007)
69. Paton, N.W., Díaz, O.: Active database systems. ACM Comput. Surv. 31, 63–103 (1999)
70. Peterson, J.L.: Petri net theory and the modeling of systems. Prentice-Hall, Englewood Cliffs (1981)
71. Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
72. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN tools for editing, simulating, and analysing coloured petri nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 450–462. Springer, Heidelberg (2003)
73. Reisig, W.: Petri Nets: An Introduction. Monographs in Theoretical Computer Science: An EATCS Series, vol. 4. Springer, Berlin (1985)
74. Reisig, W.: Petri nets with individual tokens. Theoretical Computer Science 41, 185–213 (1985)
75. Scheer, A.W.: ARIS Business Process Modelling. Springer (1999)

76. Schmidt, K.: Distributed Verification with LoLA. Fundamenta Informatica 54(2-3), 253–262 (2003)
77. Sidorova, N., Stahl, C., Trčka, N.: Workflow Soundness Revisited: Checking Correctness in the Presence of Data While Staying Conceptual. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 530–544. Springer, Heidelberg (2010)
78. Software-Ley. COSA User Manual. Software-Ley GmbH, Pullheim, Germany (1998)
79. Stahl, C.: Service Substitution. PhD thesis, Technische Universiteit Eindhoven (2009)
80. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding Substitutability of Services with Operating Guidelines. In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 172–191. Springer, Heidelberg (2009)
81. Stahl, C., Wolf, K.: Deciding service composition and substitutability using extended operating guidelines. Data & Knowledge Engineering 68(9), 819–833 (2009)
82. Stevens, W.P., Meyers, G.J., Constantine, L.L.: Structured Design. IBM Systems Journal 13(2), 115–139 (1974)
83. Suzuki, I., Kasami, T.: Three measures for synchronic dependence in petri nets. Acta Informatica 19, 325–338 (1983)
84. Szyperski, C.: Component Software – beyond Object-Oriented Programming. Addison-Wesley and ACM Press (1998)
85. van der Toorn, R.A.: Component-Based Software Design with Petri Nets - An Approach Based on Inheritance of Behavior. PhD thesis, Technische Universiteit Eindhoven (2004)
86. Valk, R., Girault, C.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer, Berlin (2003)
87. Verbeek, H.M.W., Basten, T., van der Aalst, W.M.P.: Diagnosing Workflow Processes using Woflan. The Computer Journal 44(4), 246–279 (2001)
88. van der Werf, J.M.E.M.: Compositional Design and Verification of Component-Based Information Systems. PhD thesis, Technische Universiteit Eindhoven (2011)
89. Wolf, K.: Generating Petri Net State Spaces. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 29–42. Springer, Heidelberg (2007)
90. Wolf, K.: Does my service have partners? In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 152–171. Springer, Heidelberg (2009)
91. Workflow Management Coalition. Workflow management Coalition Terminology and Glossary. Technical Report Document Number WMFC-TC-1011 – issue 3.0, Workflow Management Coalition (October 2002)
92. Workflow Management Coalition. Workflow Process Definition Interface – XML Process Definition Language, Document Number WMFC-TC-1025 – 1.0 final draft (October 2002)

Structure Theory of Petri Nets

Eike Best[1] and Harro Wimmel[2]

[1] Parallel Systems, Department of Computing Science
Carl von Ossietzky Universität Oldenburg, D-26111 Oldenburg, Germany
eike.best@informatik.uni-oldenburg.de
[2] Theory of Programming Languages and Programming,
Institut für Informatik
Universität Rostock, Albert-Einstein-Straße 22, D-18059 Rostock, Germany
harro.wimmel@uni-rostock.de

Abstract. The aim of this tutorial is to give a concise, but nonetheless not too narrow, overview of definitions and results pertaining centrally to Petri net structure theory. The Petri net model considered in these notes are the classical place/transition nets, as they have been defined in the First Advanced Course held in 1979 and originate back to the late Sixties. Structure theory asks what behavioural properties of a Petri net can be derived from its structural properties. Other aspects of Petri nets are neglected to a large extent in the present notes, such as various extensions and generalisations of central notions and results, as well as almost all algorithmic and complexity-theoretic consequences that accompany the structure-theoretic results. Because full proofs can easily be retrieved from the literature, they are not given, unless they are small and perhaps somewhat characteristic for Petri net oriented reasoning. Proof ideas are often sketched, however, and the sharpness of various results is accentuated by means of examples and counterexamples. A list for further reading is also provided.

1 First Steps in Petri Nets

We all may remember a lecture in Theoretical Computer Science in which Finite Automata were introduced. Finite Automata may be used to represent regular languages. Other classes of languages were also introduced and analysed. This was done for good reasons. For instance, compiler construction is grounded on a variety of language types.

However, we may also view formal languages from a more system-oriented perspective. If we interpret every letter as an atomic activity, then the words of a language describe the sequences of actions that are feasible. For instance, the evolutions permitted in some industrial production process could be described in this way. The atomic actions could perhaps be ascribed to the activities of various machines involved in the process. This idea can be exploited both for the simulation and for the validation of a production process in its planning stage, before it is actually implemented. Much money can be saved if design errors are detected and corrected in this way, well before the physical realisation

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 162–224, 2013.

of a production process. However, it must be realised that in general, and in particular in such processes, more than just one activity can be executed in parallel. Such parallelism (or concurrency) cannot be described either by a formal language or by a finite automaton, at least not in the form in which they are usually taught in a beginners' course. Problems concerning parallelism cannot be handled just by using words built as sequences of the letters of an alphabet.

In the beginning of the Sixties of the last century, Carl Adam Petri became aware of this lack of descriptive power of the traditional finite automata model, and of its bias towards sequential rather than concurrent execution. His dissertation, which he completed in the year 1962, was fundamentally concerned with redressing this balance. The idea behind Petri nets (as they were called some years later) is to modify some of the concepts behind finite automata. Most fundamentally, in his view, states are thought to be structured and may consist of smaller parts (called local states). Transitions may affect certain local states but may leave other local states unchanged or unaffected. Local states are represented in Petri nets by means of *places* while state transitions are again simply called *transitions*. It is this principle of locality, together with the duality between states and transitions, which underlies the very definition of a Petri net.

1.1 Basic Definitions

We shall use standard, though not always unique, mathematical notation. For instance, $f\colon X \to Y$ and $f \in Y^X$ both denote the fact that f is a function from X to Y.

Definition 1. *Petri net*
A Petri net is a triple (S, T, F) consisting of

- a countable set S of *places* and a countable set T of *transitions* with $S \cap T = \emptyset$,
- and a mapping $F\colon (S \times T) \cup (T \times S) \to \mathbb{N}$ which defines *arcs* (also called *arrows*, or *edges*) between places and transitions. $F(s,t)$ defines the number of arcs from s to t. Analogously, $F(t,s)$ defines the number of arcs from t to s. ■

In the following, we will almost exclusively consider *finite* Petri nets, that is, Petri nets in which both the set of places and the set of transitions are finite. For such nets, we often use finite sets of indices as follows: $S = \{s_1, \ldots, s_{|S|}\}$ and $T = \{t_1, \ldots, t_{|T|}\}$. Sometimes transitions are simply denoted $\{a, b, c, \ldots\}$. This is to be understood such that a is t_1, b is t_2, etc.

In the graphical representation of a Petri net, we draw every place as a circle and every transition as a box (normally square, and in general, rectangular). Furthermore, we draw exactly $F(s_i, t_j)$ arcs from the ith place s_i to the jth transition t_j, and $F(t_j, s_i)$ arcs from the jth transition t_j to the ith place s_i.

Places and transitions are enumerated for reasons of convenience, but in general, any naming is allowed. Enumerations are helpful for an alternative representation of the arcs in the calculus of matrices. In this representation,

we replace the mapping F by two $|S| \times |T|$-matrices $\mathbb{F}, \mathbb{B} \in \mathbb{N}^{S \times T}$. The value $\mathbb{B}_{i,j}$ in the ith row and jth column is, by definition, the number of arcs from s_i to t_j, and the value $\mathbb{F}_{i,j}$ is, by definition, the number of arcs from t_j to s_i. We will also occasionally write $\mathbb{F}(s)$ or $\mathbb{F}(s,.)$ for the 'sth row' in $\mathbb{F}$, given $s \in S$, $\mathbb{F}(t)$ or $\mathbb{F}(.,t)$ for the 'tth column', given $t \in T$, and $\mathbb{F}(s,t)$ for the entry in row s and column t of $\mathbb{F}$. $\mathbb{B}$ and $\mathbb{F}$ are called *backward matrix* and *forward matrix*, respectively. This is to be understood from the point of view of transitions: arcs emanating from a transition (i.e. 'forward arcs', as seen from this transition) are described by the forward matrix. Looking backward from a transition, one encounters its incoming arcs which are described in the backward matrix. As we will see later, these matrices allow linear algebra to be applied.

We introduce a number of elementary concepts.

Definition 2. *Preset, postset, and related notions*
Let (S,T,F) be a Petri net. For $x \in S \cup T$, we call ${}^\bullet x = \{y \in S \cup T \mid F(y,x) \geq 1\}$ and $x^\bullet = \{y \in S \cup T \mid F(x,y) \geq 1\}$ the *preset* (*postset*, respectively) of x. Generalising this, we define ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ and $X^\bullet = \bigcup_{x \in X} x^\bullet$, for $X \subseteq S \cup T$. An element $x \in S \cup T$ satisfying ${}^\bullet x \cup x^\bullet = \emptyset$ is called *isolated*. If there are arcs in both directions between a place s and a transition t, i.e. if $F(s,t) \geq 1 \leq F(t,s)$, then this situation is called a *loop* or a *self-loop*. A loop is called *simple* if $F(s,t) = 1 = F(t,s)$. A net is *pure* if there are no self-loops, and *plain* if there are no multiple arcs, that is, if the function F returns 0 or 1, but no number greater than 1. ■

Definition 3. *States and markings*
Let $N = (S,T,F)$ be a Petri net. The *state set* of N is defined to be $\mathbb{N}^S$, that is, the set of all functions from S to $\mathbb{N}$. This is to be understood as the set of all *potential* states of N, of which the actually possible states – to be defined later – are a subset. A function $M\colon S \to \mathbb{N}$ is called a *state*, or a *marking*, of N. If $M(s_i) = m$ then we say that 'place s_i carries m tokens (in the state M)'. We often write states as column vectors, indexed by places. ■

Graphically, we represent tokens as solid dots within a place.

Using the matrix representation of arcs, the Petri net shown in Figure 1 can be described by the quadruple $(S,T,\mathbb{B},\mathbb{F})$ and the state M with

$$S=\{s_1,s_2,s_3\},\quad T=\{t_1,t_2,t_3\},\quad \mathbb{B}=\begin{pmatrix} t_1 & t_2 & t_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{matrix} s_1 \\ s_2 \\ s_3 \end{matrix},\quad \mathbb{F}=\begin{pmatrix} t_1 & t_2 & t_3 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 0 \end{pmatrix}\begin{matrix} s_1 \\ s_2 \\ s_3 \end{matrix},\quad M=\begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}\begin{matrix} s_1 \\ s_2 \\ s_3 \end{matrix}$$

We have ${}^\bullet s_1 = \{t_1,t_3\}$, $s_1{}^\bullet = \{t_1\}$, ${}^\bullet s_2 = \emptyset$, $s_2{}^\bullet = \{t_2\}$, ${}^\bullet s_3 = \{t_1,t_2\}$, $s_3{}^\bullet = \{t_3\}$, ${}^\bullet t_1 = \{s_1\}$, $t_1{}^\bullet = \{s_1,s_3\}$, ${}^\bullet t_2 = \{s_2\}$, $t_2{}^\bullet = \{s_3\}$, ${}^\bullet t_3 = \{s_3\}$, $t_3{}^\bullet = \{s_1\}$. The function F can also be written down, but this would be rather circumstantial, compared with the graphical representation.

This Petri net has a loop between s_1 and t_1, as well as a *multiple arc* leading from t_2 to s_3; in other words, we have $F(t_2,s_3) > 1$. A multiple arc will often be

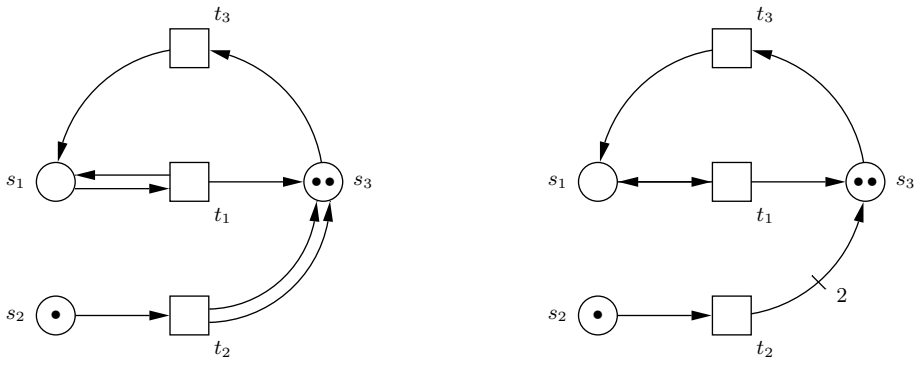


Fig. 1. Two representations of the same Petri net

represented as a plain arc together with some natural number which is inscribed at it. This number denotes the multiplicity of the arc. If the multiplicity is 1, we simply omit the number. A loop may also be represented by a simple arc with two arrow heads leading in both directions. (However, this representation can be problematic in small figures where loops could easily be mistaken for plain arcs.)

Definition 4. *Petri net with an initial marking*
An *initial Petri net* N is defined to be a tuple $N = (S, T, \mathbb{B}, \mathbb{F}, M_0)$, or $N = (S, T, F, M_0)$, where

- $(S, T, \mathbb{B}, \mathbb{F})$ (respectively, (S, T, F)) is a Petri net;
- $M_0 \in \mathbb{N}^S$ is an *initial state*.

We also denote a Petri net $N = (S, T, F)$ with initial state M_0 by (N, M_0). The initial state is also called *initial marking* or *starting state*. ■

In the following, we will call an initial Petri net simply a Petri net, if it is clear from the context that we are talking about an initially marked net. Also, we often use the word *system* in an informal way when talking about an initially marked net. Typically, the starting state is included in the graphical representation of a net by means of tokens (solid dots) on the places. In Figure 1, for example, $M = (0 \quad 1 \quad 2)^\mathsf{T}$ is the starting state. The notation $^\mathsf{T}$ means 'transposed' and shows, in this case, that the marking is viewed as a column vector.

Sometimes it is desirable to ignore certain parts of a Petri net, for instance a set of transitions and/or places, together with all arcs connected to them. This leads to the notion of a *subnet*.

Definition 5. *Subnet*
Let $N = (S, T, F, M_0)$ be a Petri net and let $S' \subseteq S$ as well as $T' \subseteq T$. The *subnet induced by* S' *and* T' is denoted by $N(S', T')$ and defined by

$$N(S', T') = (S', T', F|_{(S' \times T') \cup (T' \times S')}, M_0|_{S'}).$$

Generally, the vertical bar denotes the restriction of the domain of a set or function to the set given in its index. In this particular case, we have $F|_{(S' \times T') \cup (T' \times S')} = F \cap (((S' \times T') \cup (T' \times S')) \times \mathbb{N})$. ■

The idea is that in $N(S', T')$, the part of the net defined by S' and T' is left intact, while all elements of $S \setminus S'$ and $T \setminus T'$ are neglected. The definition of F' means that all arcs between elements of S' and T' (including their multiplicities) are just inherited from N to $N(S', T')$. All other arcs, that is, arcs with at least one endpoint in $S \setminus S'$ or in $T \setminus T'$, are ignored. The definition of $M_0|_{S'}$ means that places in S' carry exactly as many tokens in $N(S', T')$ as they do in N. Places in $S \setminus S'$ and all tokens on them are ignored.

In Figure 1, the subnet induced by $S' = \{s_1\}$ and $T' = \{t_1, t_2, t_3\}$ consists of one place, s_1, with zero tokens, three arcs (two of which form a loop), and three transitions (one of which, viz. t_2, is isolated).

1.2 Firing Transitions

We now describe the dynamics, i.e. the behaviour, of initially marked nets. This will be defined in analogy to the successive execution of state transitions in a finite automaton. In Petri nets, this process is called *firing* or *executing* transitions.

Definition 6. *The transition rule of Petri nets*
Let $N = (S, T, \mathbb{B}, \mathbb{F})$ be a Petri net, let $M \in \mathbb{N}^S$ be a state of N, and let $t \in T$ be a transition of N. We call t *M-activated* (or *enabled*, *firable*, *executable* in state M), if $M \geq \mathbb{B}(t)$ (that is, $\forall s \in S\colon M(s) \geq \mathbb{B}_{s,t} = F(s,t)$).
A transition t *fires in state M to state M'* (or *is executed* in state M, leading to state M', or simply *leads from M to M'*) if:

- $M \geq \mathbb{B}(t)$ (that is, t is activated in M), and
- $M' = M - \mathbb{B}(t) + \mathbb{F}(t)$.

This rule is called the *transition (or firing) rule*, and it is the basic behavioural (state change) rule for Petri nets. Formally, the fact that t is firable in M and leads from M to M' is denoted by $M\,[t\rangle M'$ or by $M \xrightarrow{t} M'$. ■

Informally, when a transition fires, it consumes tokens from every place of its preset (whence there needs to be at least one token on every such place just prior to firing) and produces tokens on every place of its postset. The number of tokens consumed and produced are calculated according to the multiplicity of arcs around the transition. More precisely, if a place s has an outgoing arc of multiplicity k towards t, then every single firing of t needs at least k tokens on s and consumes exactly k tokens from s. Similarly, if t is connected to a place s' of its postset by an arc of multiplicity k, then every single firing of t produces exactly k tokens on s', which are added to the already existing ones. In the case of self-loops, tokens are first taken from a place and later reproduced. That is, if $F(s'', t) = k > 0$ and $F(t, s'') = m$, k tokens on s'' are necessary for firing t. When firing t, we might think of it as the k tokens being removed from s'' first,

and then, in a second step, m tokens being added to s'' again. In the special case of a simple loop ($k = m = 1$), the effect of firing is that the number of tokens on such a place is neither decreased nor increased, because the single token that is taken away by firing, is put back by the same firing.

As an example, let us reconsider the Petri net from Figure 1. We have, on the one hand, that

$$(0\ \ 1\ \ 2)^\mathsf{T}\,[t_2\rangle(0\ \ 0\ \ 4)^\mathsf{T} \quad \text{and} \quad (0\ \ 1\ \ 2)^\mathsf{T}\,[t_3\rangle(1\ \ 1\ \ 1)^\mathsf{T}.$$

On the other hand, firing t_1 in state $(0\ \ 1\ \ 2)^\mathsf{T}$ is not possible, since there is no token on s_1. We may also fire from other states. For instance, we have $(7\ \ 3\ \ 5)^\mathsf{T}\,[t_1\rangle(7\ \ 3\ \ 6)^\mathsf{T}$ and $(2\ \ 1\ \ 0)^\mathsf{T}\,[t_2\rangle(2\ \ 0\ \ 2)^\mathsf{T}$.

Of course, it is usually possible to execute a sequence of transitions, one after another, instead of just one of them. This naturally leads to two interesting questions:

- How can the set of states that are reachable from the initial state through such sequences be characterised?
- How can the set of firable sequences be characterised?

As we will see, these two questions are of a rather different nature. Moreover, the answers to both of them are non-trivial.

Definition 7. *Firing sequence*
Let $N = (S, T, F, M_0)$ be a Petri net. We define inductively: $\forall t \in T,\ \sigma \in T^*$:

$$M\,[\varepsilon\rangle M' \ \text{ if } \ M = M'$$
$$M\,[\sigma t\rangle M' \ \text{ if } \ \exists M'' \in \mathbb{N}^S :\ M\,[\sigma\rangle M''\,[t\rangle M',$$

where $M\,[\sigma\rangle M''\,[t\rangle M'$ is a shorthand notation for $M\,[\sigma\rangle M'' \wedge M''\,[t\rangle M'$ and ε is the empty sequence.
We read $M\,[\sigma\rangle M'$ as 'σ fires from M to M'', or 'σ is executed from M and leads to M'', or, more simply, 'σ leads from M to M''. Here also, $M \xrightarrow{\sigma} M'$ is synonymous to $M\,[\sigma\rangle M'$.

$$M\,[\sigma\rangle \quad :\Longleftrightarrow \quad \exists M' \in \mathbb{N}^S :\ M\,[\sigma\rangle M'.$$

A sequence $\sigma \in T^*$ is called a *firing sequence* or an *execution (sequence)* from M (or *executable/firable at* M), denoted by $M\,[\sigma\rangle$, if there is some marking M' with $M\,[\sigma\rangle M'$.
Further, we call $\mathcal{E}(M) = \{M' \mid \exists \sigma \in T^*\colon M\,[\sigma\rangle M'\}$ the *reachability set* (or the *state space*) of M, and $\mathcal{E}(N) := \mathcal{E}(M_0)$ is the reachability set of N. Alternatively, we also write $[M\rangle$ instead of $\mathcal{E}(M)$. ■

In Figure 1, we have the following firing sequences: $\sigma_1 = t_3t_3t_1t_3t_2t_3t_3$, or $\sigma_2 = t_3t_1t_1t_1t_3$, amongst others. More precisely, we have $(0\ \ 1\ \ 2)^\mathsf{T} \xrightarrow{\sigma_1} (5\ \ 0\ \ 0)^\mathsf{T}$ and $(0\ \ 1\ \ 2)^\mathsf{T} \xrightarrow{\sigma_2} (2\ \ 1\ \ 4)^\mathsf{T}$. By contrast, $\sigma_3 = t_3t_3t_1t_3t_3t_2t_3$ is not firable from $M_0 = (0\ \ 1\ \ 2)^\mathsf{T}$, since the fourth instance of t_3 cannot be executed.

The reachability set, $\mathcal{E}(M_0)$, of this example is as follows:

$$\{(0\ \ 1\ \ 2)^\mathsf{T}, (0\ \ 0\ \ 4)^\mathsf{T}\} \cup \{(i\ \ j\ \ k)^\mathsf{T} \mid i \geq 1 \wedge j \leq 1 \wedge i + 2j + k \geq 4\}. \qquad (1)$$

The reachability set does not have to be representable so smoothly, even if the given Petri net is small.

Firing enjoys several basic properties that one should know about. The state reached after firing a transition depends only on the previous state, rather than on the (possibly large) history by which this state has been reached; this property is called *memorylessness* or *local determinacy* of firing. If a transition can be fired in some state, then it can also be fired in every 'larger' state (where larger means that no place contains less tokens and at least one place more tokens); this property is called the *monotonicity* of firing. If two transitions can be fired in arbitrary order, then the resulting marking after firing both does not depend on their ordering; this is called the *commutativity* of firing. More formally, we have:

Lemma 1. *Properties of firing*
Transition firing is
- *locally determined*, i.e. $\forall t \in T\ \forall M, M', M'' \in \mathbb{N}^S$: $(M[t\rangle M' \wedge M[t\rangle M'' \Rightarrow M' = M'')$,
- *monotonic*, i.e. $\forall M, M', M'' \in \mathbb{N}^S\ \forall t \in T$: $(M\,[t\rangle M' \Longrightarrow (M + M'')\,[t\rangle (M' + M''))$,
- and *commutative*, i.e. if M, M_1, M_2, M_3, M_4 are states and t, t' are transitions with $M\,[t\rangle M_1\,[t'\rangle M_2$ and $M\,[t'\rangle M_3\,[t\rangle M_4$, then $M_2 = M_4$.

Proof: Local determinacy: We have $M' = M - \mathbb{B}(t) + \mathbb{F}(t) = M''$.
Monotonicity: $M \geq \mathbb{B}(t)$ entails $M + M'' \geq \mathbb{B}(t) + M'' \geq \mathbb{B}(t)$ and $M' + M'' = M - \mathbb{B}(t) + \mathbb{F}(t) + M'' = (M + M'') - \mathbb{B}(t) + \mathbb{F}(t)$.
Commutativity: $M_2 = M_1 - \mathbb{B}(t') + \mathbb{F}(t') = M - \mathbb{B}(t) + \mathbb{F}(t) - \mathbb{B}(t') + \mathbb{F}(t') = M - \mathbb{B}(t') + \mathbb{F}(t') - \mathbb{B}(t) + \mathbb{F}(t) = M_3 - \mathbb{B}(t) + \mathbb{F}(t) = M_4$. ■

It is important to note that commutativity does not mean that $t't$ is firable whenever tt' is. Thus, the above concept of commutativity is not exactly the same as that used frequently in mathematics. Mathematical commutativity is not normally satisfied in Petri net firings. More precisely, the following properties are *not* normally satisfied:
- *persistency*, i.e. $\forall t, t' \in T\ \forall M \in \mathbb{N}^S$: $(t \neq t' \wedge M\,[t\rangle \wedge M\,[t'\rangle \Longrightarrow M\,[tt'\rangle)$.
- *confluence*, i.e. $\forall M, M', M'' \in \mathbb{N}^S\ \forall \sigma, \sigma' \in T^*$: $(M\,[\sigma\rangle M' \wedge M\,[\sigma'\rangle M'' \Longrightarrow \exists \widehat{M} \in \mathbb{N}^S\ \exists \sigma'', \sigma''' \in T^*$: $(M'\,[\sigma''\rangle \widehat{M} \wedge M''\,[\sigma'''\rangle \widehat{M}))$.

Both properties can be disproved by the following simple Petri net:

(2)

Persistency means that an activated transition cannot be de-activated by *other* transitions. In general, however, it is possible for some transition to de-activate another one; such a situation is called a *conflict*. In (2), for example, t_2 is activated in marking M. However, firing t_1 de-activates t_2 (as well as t_1 itself). In section 4, persistency and the absence of conflicts will be investigated more closely.

Confluence means that executions drifting apart in a net can be brought together again. In general, however, this may not be the case. In (2), the markings M_1 and M_2 reached after firing t_1 and t_2 from M, respectively, have no common successor marking. For example, if t_2 denoted an erroneous exection, then there would be no way of 'correcting' the error by reaching a state that could also have been reached after t_1.

1.3 Graphs and Multigraphs

The graphical representation of a Petri net indicates that it can be understood as a *graph* in the mathematical sense. In this section, we recall some notions pertaining to (multi-)graphs in general.

A *graph* is a structure (X, E) where $E \subseteq X \times X$. An element of X is called a *vertex*, or a *node*. An element $e = (x, y) \in E$ is called an *arc*, or an *edge*, or sometimes also an *arrow*, leading from x to y. An *arc-labelled* graph is a structure (X, L, E) where $E \subseteq X \times L \times X$. For $e = (x, \ell, y) \in E$, ℓ is also called the *label* (or the *inscription*) of the arc e from x to y. A *multigraph* is a structure $(X, \mathbb{E})$ where $\mathbb{E}$ is a multiset of pairs from $X \times X$. Thus, we may have several arrows in parallel from one node to another one. Likewise, a *labelled multigraph* is a structure $(X, L, \mathbb{E})$ where $\mathbb{E}$ is a multiset of triples from $X \times L \times X$.

A *subgraph* of a (multi-)graph with vertex set X is a graph with vertex set $X' \subseteq X$ and (labelled) arcs restricted to those between nodes in X'. By a (directed) *path* we mean a directed sequence of edges, such that the endpoint of one edge is the beginning of the next one. A path is a *cycle* if its starting vertex equals its end vertex. The *length* of a path is the number of edges in it. A special case is just a single node without any edge (called a path of length 0). A path is called *simple* or *elementary* if no vertex appears twice in it, except possibly the very beginning and the very end, in which case it is called a *simple cycle* or an *elementary cycle*.

A (multi-)graph G is called *strongly connected* if for any two nodes x and y, there is a directed path from x to y. G is called *weakly connected* if for any two nodes x, y, there is some sequence of arrows (not necessarily pointing in the same direction) from x to y. G is called *covered by (directed) cycles* if for any arrow from x to y, there is a directed path from y to x. It is clear that if a graph is strongly connected, then it is also weakly connected and covered by cycles. Conversely, if a graph is covered by cycles and weakly connected, then it is also strongly connected. A *strongly connected component* (*weakly connected component*) of a graph G is a maximal subset X of vertices such that the subgraph G' with vertex set X is strongly (weakly) connected.

1.4 The Reachability Graph

In this section, we investigate the set of states of a Petri net (S, T, F, M_0) that can be reached during its execution(s). The concept of *reachability set* has already been introduced. It comprises all states that can be reached after the execution of arbitrary firing sequences, including the initial state which is reached after 'firing' the empty sequence. This set can be provided with some structure. Instead of just recording the reachable states, we may also record a relation between them, namely the information which transition has to be fired in order to get from one state to another one. In this way, we obtain the *reachability graph*.

Definition 8. *The reachability graph*
Let $N = (S, T, F)$ be a Petri net, let $M \in \mathbb{N}^S$ and let $\mathcal{E}(M)$ be the reachability set of M in N. The *reachability graph* $RG(N, M)$ is defined to be an arc-labelled graph $(\mathcal{E}(M), E)$ with the following set of arcs:

$$E \;=\; \{(M_1, t, M_2) \mid M_1 \in \mathcal{E}(M) \wedge M_1\,[t\rangle M_2\}.$$

If $N = (S, T, F, M_0)$ is a Petri net with an initial marking, then the reachability graph of N, $RG(N)$, is defined to be $RG(N) = RG(N, M_0)$. ■

The reachability graph is always weakly, but not necessarily strongly, connected. Ignoring the arc labelling, it is also a multigraph, because $M_1\,[t\rangle M_2$ and $M_1\,[t'\rangle M_2$ does not necessarily imply $t = t'$. Thus there may be two or more arrows leading from M_1 to M_2.

Let us consider two examples. The Petri net shown in Figure 2 is a modification of the net shown previously in Figure 1. The loop between s_1 and t_1 was replaced by a single arc. The modified net has the reachability graph shown in Figure 3. It has two strongly connected components and three edges that are not covered by cycles.

If we designate $(0 \quad 1 \quad 2)^{\mathsf{T}}$ as a start state and define some additional "accepting" states, the Petri net can be viewed as an automaton that accepts the

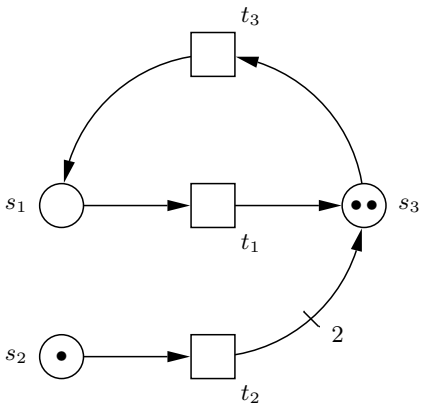


Fig. 2. A modification of Figure 1

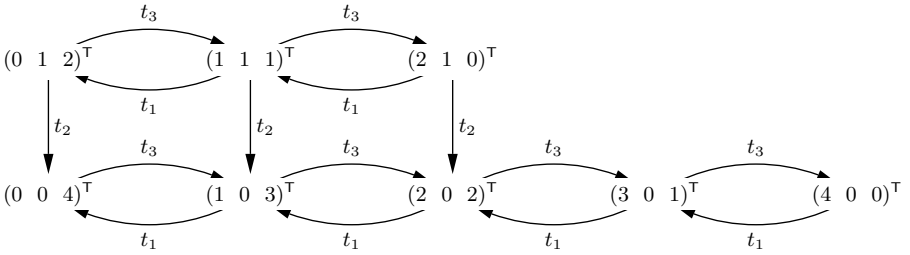


Fig. 3. The reachability graph of the Petri net shown in Figure 2; the initial marking is $(0\ \ 1\ \ 2)^\mathsf{T}$

language of all firing sequences ending in such an accepting state. With these additions we can study *Petri net languages*; such a study is, however, beyond the scope of the present notes. Suffice it here to say that Petri nets can accept non-regular languages, while, e.g., all languages of finite automata are regular. Consequently, while the reachability graph in our example looks quite similar to such finite automata, there must also be some Petri nets whose reachability graphs cannot be viewed that way. This is indeed the case.

Consider the net which was originally shown in Figure 1. It has a loop (instead of just a single arc) between s_1 and t_1, and its reachability graph $RG(N, M_0)$ with $M_0 = (0\ \ 1\ \ 2)^\mathsf{T}$ is shown in Figure 4. This reachability graph is infinite and can thus not be included fully in the Figure. Its representation in Figure 4 ends at some arbitrarily chosen (sufficiently early) point. The targets of the arcs at the bottom of the figure are not shown explicitly. This graph has infinitely many strongly connected components (each node being one).

An infinitely large reachabilibity graph is, of course, not a finite automaton. Also, in an infinite graph, the task of searching whether a given state is reachable is burdensome. Nevertheless, we may discover quickly that in our example,

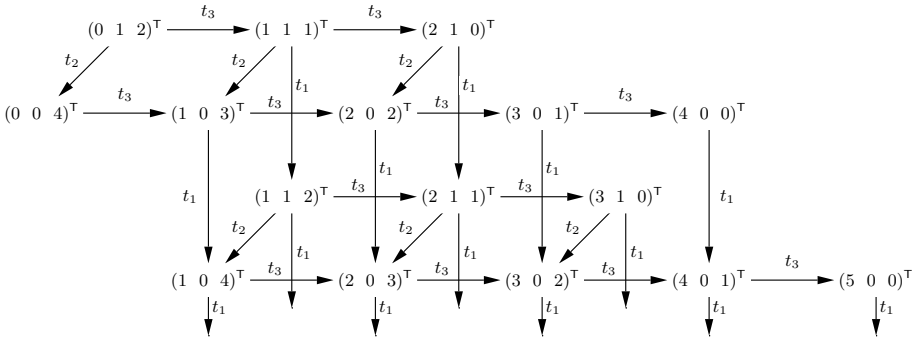


Fig. 4. Part of the reachability graph of the net shown in Figure 1; the initial state is $(0\ \ 1\ \ 2)^\mathsf{T}$

for instance, state $(1\ \ 0\ \ 9)^\mathsf{T}$ is reachable while state $(0\ \ 0\ \ 9)^\mathsf{T}$ is not. It suffices to examine the systematic structure of the graph in order to answer this or similar questions. In general, however, reachability graphs are much more complex, and the question whether a given state is reachable is very hard to answer. In fact, it has been an open question for several years whether or not this question is decidable. We cite the known result next.

Theorem 1. *Reachability is decidable*
The reachability problem, defined by

$$\mathsf{RP} \;=\; \{\ (N, M, M') \mid N = (S, T, F) \text{ is a Petri net, } M, M' \in \mathbb{N}^S, \text{ and } M' \in \mathcal{E}(M)\ \},$$

is decidable. ■

1.5 Boundedness, Safeness, Liveness, Deadlock-Freeness, and Reversibility

The reachability graph reveals more information than just the set of reachable states. We discuss some interesting and relevant properties that can be inferred from the reachability graph.

Definition 9. *Boundedness and safeness*
Let $N = (S, T, F, M_0)$ be a Petri net. A place $s \in S$ is called *safe* if $M(s) \leq 1$ whenever σ is a firing sequence and M is a state with $M_0\,[\sigma\rangle M$; s is called *m-bounded* (for $m \in \mathbb{N}$), if $M_0\,[\sigma\rangle M$ always entails $M(s) \leq m$. Place s is *bounded* if it is m-bounded, for some $m \in \mathbb{N}$, otherwise it is *unbounded*.

A Petri net $N = (S, T, F, M_0)$ is called safe (bounded) if all places $s \in S$ are safe (bounded, respectively). N is called m-bounded, for some $m \in \mathbb{N}$, if every place $s \in S$ is m-bounded. N is called unbounded if it contains an unbounded place. ■

The safeness (m-boundedness) of N can be deduced by inspecting the reachability graph $RG(N)$. Similarly, the safeness (m-boundedness) of any place can be deduced by inspection. We simply need to check all states which occur in $RG(N)$. This is practical only if $RG(N)$ is finite (and is, even then, likely to be extremely time-consuming).

Definition 10. *Liveness, deadlock-freeness, and reversibility*
Let $N = (S, T, F, M_0)$ be a Petri net.

- A transition $t \in T$ is called *singly live* or *not dead*, if there is a firing sequence σ with $M_0\,[\sigma t\rangle$.
- A transition t is called *weakly live*, if there is an infinite word $w \in T^\infty$ such that t occurs infinitely often in w and $M_0\,[w\rangle$ (meaning that $M_0\,[\sigma\rangle$ holds for every finite prefix σ of w).

- A transition t is called *live* or *strongly live*, if for every reachable marking $M \in \mathcal{E}(M_0)$, there is some firing sequence σ with $M\,[\sigma t\rangle$.
- A transition t is called *reversible*, if for every reachable marking $M \in \mathcal{E}(M_0)$, if $M\,[t\rangle M'$, then $M \in \mathcal{E}(M')$.

The Petri net N is called (singly / weakly / strongly) live or reversible, if every transition in the net is (singly / weakly / strongly) live or reversible, respectively. N is called *dead* if N contains no singly live transition. A reachable marking $M \in \mathcal{E}(M_0)$ is called a *deadlock* if no transition is activated at M. N is called *deadlock-free* if there is no marking $M \in \mathcal{E}(M_0)$ which is a deadlock. ■

It is easy to check on the reachability graph whether or not a transition is singly live. If it contains some arc (M, t, M'), then every path from M_0 to M determines a firing sequence σ with $M_0\,[\sigma\rangle M$, and in state M we have $M\,[t\rangle$. That is, t is singly live if and only if such an arc occurs in the reachability graph.

If the reachability graph is finite, then it is not hard to check whether a given transition t is weakly live. If it is, then it can be fired arbitrarily often, and since the reachability graph is finite, it must contain a cycle with an arc of the form (M, t, M'). Conversely, if the reachability graph contains such a cycle, then t is weakly live, since we can fire into the cycle, and then along the cycle arbitrarily often.

Checking strong liveness of a transition t is not so easy, but it can also, in principle, be done on the reachability graph. For every marking M contained in it, it must be checked whether a path leads from M to an arc inscribed by t. For a finite reachability graph this means every *terminal* strongly connected component (i.e., with no arcs going out to other such components) must contain an edge labelled t.

Checking deadlock-freeness can be done by examining the reachability graph for vertices which have no output arc. The net is deadlock-free if and only if such vertices are absent.

Reversibility can be checked on the reachability graph as well. Recall that the reachability graph is always weakly connected. For a transition t to be reversible, each edge labelled with t must lie on some cycle (or equivalently, within some strongly connected component) of the reachability graph, and for the whole net to be reversible, the entire reachability graph must be strongly connected. Conversely, if the reachability graph is strongly connected, then the net is reversible. Note that this holds even if the initial marking is a deadlock.

The considerations of this section are subsumed as follows:

Corollary 1. *Properties that can be checked on the reachability graph*
If the reachability graph of a Petri net $N = (S, T, F, M_0)$ is finite, then there exist terminating algorithms which decide the following properties:
- whether a place is safe;
- whether a place is m-bounded;
- whether a place is bounded;
- whether a transition is dead (i.e., not singly live);
- whether a transition is weakly live;

- whether a transition is strongly live;
- whether a transition is reversible;
- and whether the net N is safe (m-bounded, bounded, dead, singly live, weakly live, live, deadlock-free, or reversible). ■

1.6 Coverability Graphs

What happens if the reachability graph is not finite? Then none of the above questions can be solved efficiently using the reachability graph. Given that the reachability graph may be infinite and therefore unmanageable, does there perhaps exist a similar construction which always leads to a finite structure from which at least *some* interesting information about safeness, boundedness and liveness can be inferred? Such a structure has indeed been invented. It is called the *coverability graph*. Such a graph (indeed, a number of such graphs with similar properties, all of them finite) can be associated with a Petri net. If the latter is bounded, the coverability graphs defined in the literature would normally coincide with the unique reachability graph.

Some information is lost if the Petri net is unbounded, since coverability graphs are finite. However, the coverability graph(s) can still be used for some, but not all, analysis of the net. For instance, the question whether a net is bounded, can be decided on the coverability graph, as can the question which, if any, places are unbounded. As another example, however, the question whether a transition (or the entire net) is live, or weakly live, cannot be decided on the coverability graph only. For weak liveness it is sufficient to additionally know the full Petri net structure which might be partially hidden in the coverability graph. For liveness we know that it is at least as hard as the reachability problem. The latter is EXPSPACE-hard, and no upper complexity bounds are known at the present time. The reachability problem can also not be decided on the coverability graph, but there exist constructions using sequences of generalised coverability graphs to solve this problem. Further details about coverability graphs are beyond the scope of this tutorial.

1.7 Structural Boundedness, Structural Liveness, and Well-Formedness

In this section, we define two notions that are related to boundedness and liveness, but pertain to nets without markings, rather than to net systems as before. Hence there is in general no single reachability graph on which they could be checked. We introduce these properties by means of Figures 5 to 7.

Consider the system shown on the left-hand side of Figure 5. It is 2-bounded but not safe. It is also not live. Its reachability graph is shown on the right-hand side of the figure. The system shown in Figure 6 is unbounded. However, it is live. The system shown in Figure 7 is safe (i.e., 1-bounded) as well as live, but it is not reversible. The reachability graph of this net is shown on the right-hand side of Figure 7. (In order to avoid writing long vectors, the reachable markings were written in a short-hand notation. They are represented as strings of place

names, where a place appears as often as there are tokens on it.) The graph is composed of two separate strongly connected components, one containing just $M_0 = s_2 s_5$ and the other containing all other markings.

Definition 11. *Structural boundedness, structural liveness, and well-formedness* A net $N = (S, T, F)$ is called

- *structurally bounded*, if for all markings M, the system (S, T, F, M) is bounded;
- *structurally live*, if there is some marking M such that the system (S, T, F, M) is live;
- *well-formed*, if there is some marking M such that the system (S, T, F, M) is bounded and live.

A marking M of a net (S, T, F) is called *live (bounded)* if the system (S, T, F, M) is live (bounded). ■

Note that there is a significant difference between the notions of structural boundedness and structural liveness: while in the first case, boundedness must hold for *all* possible initial markings, in the second case it is sufficient that there exists *some* initial marking for which liveness holds.

If we omit the three tokens from the net shown on the left-hand side of Figure 5, then we get a net which is not structurally live. To see this, we actually have to investigate not just the initial marking which is shown in the figure, but any other initial marking as well, that is, we need to consider an infinite number of reachability graphs and check that none of them belongs to a live net. Indeed, let an arbitrary initial marking be given and consider a maximal sequence that arises by choosing b instead of a whenever both are enabled. It is easy to see that such a sequence always leads to a deadlock, and thus, the net is not live. On the other hand, the net is structurally bounded. To see this, we again need an argument showing that the net is not only bounded for the marking shown in the figure, but

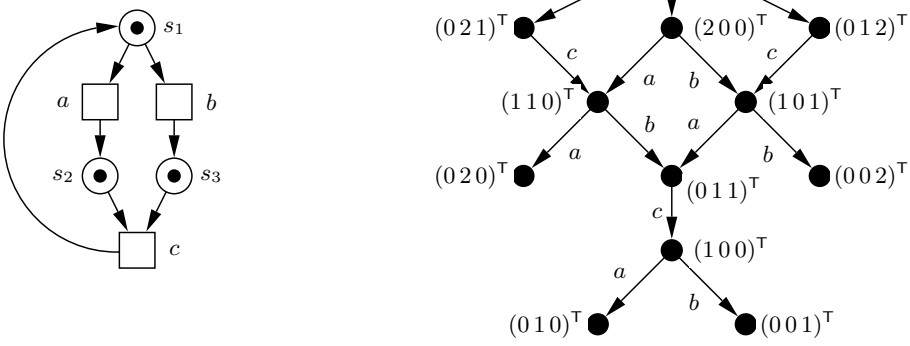


Fig. 5. Neglecting the tokens: structurally bounded, but not structurally live

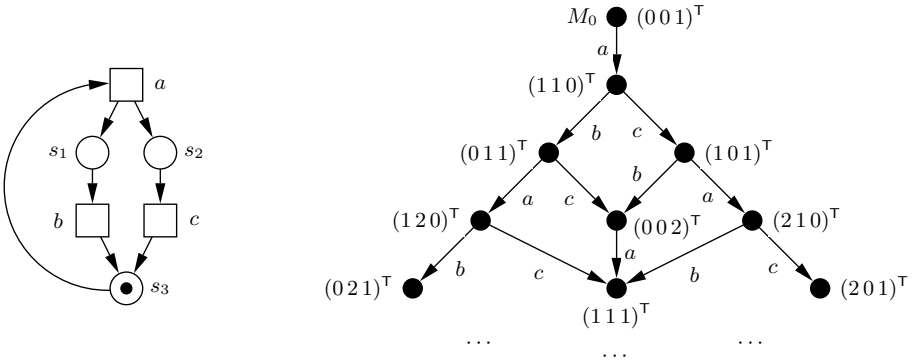


Fig. 6. Neglecting the token: structurally live, but not structurally bounded

for all other possible initial markings as well. In fact, this follows easily from the fact that for any arbitrary initial marking, the overall number of tokens of this net can never increase. Finally, this net is not well-formed, because it has no live marking and, *a fortiori*, no bounded and live marking.

The net shown on the left-hand side of Figure 6 (neglecting the token) is structurally live. To see this, it suffices to exhibit a live marking; for instance, the marking shown in the figure is indeed live. The net is, however, not structurally bounded. To see this, it suffices to exhibit a non-bounded marking; for instance, the marking shown in the figure is not bounded. The net is not well-formed, either, because its only bounded marking (which is the empty – i.e., token-free – marking) fails to be live.

The net shown on the left-hand side of Figure 7 (neglecting the two tokens) is well-formed. To see this, it suffices to exhibit a live and bounded marking; for instance, the marking shown in the figure is both live and bounded. It follows that the net is structurally live. It is not clear, at this point, whether the net is also structurally bounded. Later, we will prove that this is indeed the case.

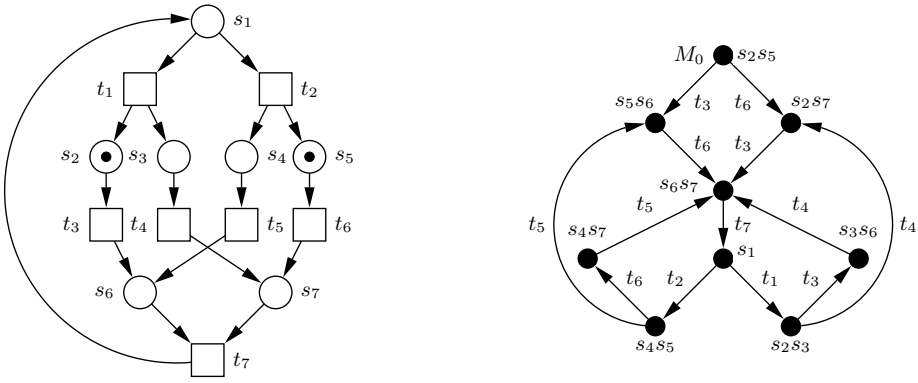


Fig. 7. Neglecting the tokens: structurally bounded and structurally live

1.8 Bibliographical Remarks and Further Reading

Petri nets were conceived in [Pet62] and brought into the form presented in these notes, called the place/transition nets, by several research teams, prominent amongst whom were Hartmann Genrich, Amir Pnueli, et al. [CHEP71, GL73]. Around the same time, a closely related model, the vector addition systems, was put forward and investigated by Richard Karp and others [KM69]. The notion of a coverability tree was defined in [KM69] to serve as the basis for an algorithm to decide boundedness. Both models have sparked several interesting developments and led to famous results, such as the decidability of reachability which was proved independently by Ernst Mayr in his dissertation [May80, May84] and by Rao Kosaraju [Kos82]. This particular result was made more widely accessible through a new proof by Jean-Luc Lambert [Lam92], by further work by Jérôme Leroux [Ler09], and to a German-speaking audience, by a textbook by Lutz Priese and Harro Wimmel [PW03]. The proof ideas which were contained in these works have proved useful in obtaining further results, such as described in [Wim04] and in [HMW10]. Even before reachability was known to be decidable, a number of properties had been shown to be reducible to reachability [Hac74].

The notions of liveness and boundedness originated, to our knowledge, from the early work cited above [KM69, CHEP71, GL73]. Soon after these publications, Leslie Lamport coined the terminology of liveness versus safety properties in connection with program verification [Lam77]. He later said that he used these terms based on a slight misunderstanding of the similar terms coming from Petri net theory [Lam]. As readers proficient in verification will surely be aware of, they have since led a very meaningful and independent scientific life as well.

There are several textbooks and overview articles on place/transition nets, e.g. by Wolfgang Reisig [Rei85], James Lyle Peterson [Pet81] and Tadao Murata [Mur89]. The reader is also referred to the bibliography entries mentioned in `http://www.informatik.uni-hamburg.de/TGI/PetriNets/bibliographies/`

2 Linear-Algebraic Structure of Petri Nets

Both a Petri net N and its reachability graph $RG(N)$ are mathematical structures known as *(multi-)graphs*. However, the graph-theoretical structure of N bears no apparent relationship to the graph-theoretical structure of $RG(N)$. It may be the case that N is a very complicated graph and that $RG(N)$ is extremely simple, but it may also be the case that $RG(N)$ is very complex even though N looks rather innocent.

Normally, there is a large discrepancy between the size of N and the size of $RG(N)$. While N is of the order of a computer program's size (which may vary between a few and several hundreds of millions of lines), $RG(N)$ is often exponentially larger than N. Therefore, it has long been one of the objectives of net theory to be able to deduce some properties of $RG(N)$ from properties of N itself. It is particularly interesting to find results that would allow one to check properties such as those of Corollary 1 (e.g., boundedness, or liveness) by checking the structure of the net only, without constructing the reachability

graph (or the coverability graph). This idea has been known by the name of *structure theory*, or the *structural analysis*, of Petri nets.

Structure theory will be investigated from different points of view.

In the present section, we will exploit the *linear-algebraic* structure of N in order to deduce properties of its behaviour under a given initial marking. In section 3, by contrast, we will exploit the *graph-theoretical* structure of N for the same purpose. The final section 4 reveals some connections between net properties that are defined partly structurally and partly behaviourally. In most cases, the objective is to use *static properties* of the net N, i.e., properties that can be 'read off' its structure, in order to deduce *dynamic properties*, i.e. properties such as boundedness or liveness or the absence of conflicts.

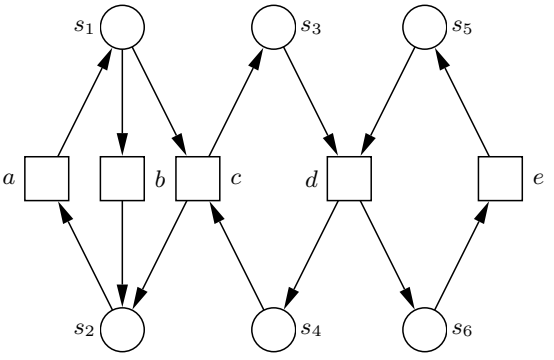
2.1 Incidence Matrix, Marking Equation, Marking Inequality, and Realisability

The incidence matrix of a Petri net is derived from the two matrices $\mathbb{F}$ and $\mathbb{B}$, and it forms the basis for linear-algebraic manipulations of Petri nets. An example is shown in Figure 8.

Definition 12. *Incidence matrix*
Let $N = (S, T, F)$ be a net. The *incidence matrix*, or *connectivity matrix*, of N is defined as the function $C\colon S \times T \to \mathbb{Z}$ with $C = \mathbb{F} - \mathbb{B}$. ■

> If C has no rows, or no columns, or both, linear algebra cannot reasonably be expected to work. Therefore, we will assume that there is at least one transition and at least one place in the nets we consider.

Incidence matrices do not capture loops. In general, N cannot be reconstructed uniquely from C, because some information about loops is lost. For example, consider the net which consists only of a place s and a transition t, without any arrows.



C	a	b	c	d	e
s_1	1	−1	−1	0	0
s_2	−1	1	1	0	0
s_3	0	0	1	−1	0
s_4	0	0	−1	1	0
s_5	0	0	0	−1	1
s_6	0	0	0	1	−1

Fig. 8. An unmarked net (l.h.s.) and its incidence matrix (r.h.s.)

This net has exactly the same incidence matrix as the net that consists of s and t and has two additional arrows, one from s to t and another one from t to s.

Using the incidence matrix, the marking reached after a firable sequence can be computed linear-algebraically. To see how this can be done, we need to define Parikh vectors.

Definition 13. *Parikh vector*
Let $\tau = t_1 \dots t_k \in T^*$ be a sequence of transitions from T. Let $\#(t, \tau)$ denote the number of times transition t occurs in τ. The *Parikh vector* or *occurrence count vector* of τ is defined as a (column) T-*vector* (that is, a T-based column vector) $\mathcal{P}(\tau) \in \mathbb{N}^T$ which contains, at entry t, the occurrence count $\#(t, \tau)$. ■

For example, for four transitions $\{t_1, t_2, t_3, t_4\}$,

$$\mathcal{P}(\varepsilon) = (0\,0\,0\,0)^{\mathsf{T}}, \quad \mathcal{P}(t_2) = (0\,1\,0\,0)^{\mathsf{T}}, \quad \text{and } \mathcal{P}(t_1 t_2 t_4 t_2 t_3 t_4) = (1\,2\,1\,2)^{\mathsf{T}}.$$

Comparing Definition 12 with the firing rule (section 1), it can be seen that an entry $C(s,t)$ indicates how the token count on s changes through the firing of t. That is, if $M_1\,[t\rangle M_2$, then $M_2 = M_1 + C \cdot \mathcal{P}(t)$ $(= M_1 + (\mathbb{F} - \mathbb{B})(\mathcal{P}(t)))$. For example, consider the net shown in Figure 9. The initial marking can be represented as a column vector $M_0 = (1\,0\,0\,2\,1\,0)^{\mathsf{T}}$. Transition c can fire, and we easily check that indeed,

$$\underbrace{\begin{pmatrix}0\\1\\1\\1\\1\\0\end{pmatrix}}_{M} = \underbrace{\begin{pmatrix}1\\0\\0\\2\\1\\0\end{pmatrix}}_{M_0} + \underbrace{\begin{pmatrix}1&-1&-1&0&0\\-1&1&1&0&0\\0&0&1&-1&0\\0&0&-1&1&0\\0&0&0&-1&1\\0&0&0&1&-1\end{pmatrix}}_{C} \cdot \underbrace{\begin{pmatrix}0\\0\\1\\0\\0\end{pmatrix}}_{\mathcal{P}(c)}$$

This idea can be generalised as follows.

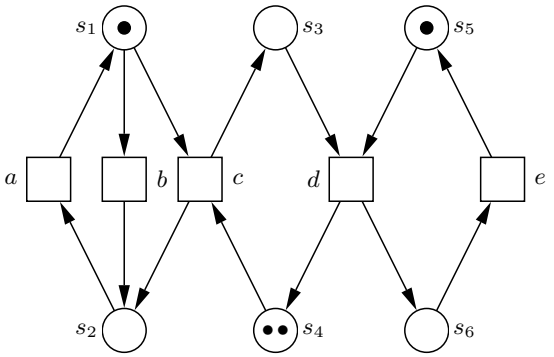


Fig. 9. The net of Figure 8 with an initial marking M_0

Lemma 2. *Firing lemma and marking equation*
If $M_1[\tau\rangle M_2$, then $M_2 = M_1 + C \cdot \mathcal{P}(\tau)$.

Proof: The claim follows easily by induction on the length of τ. ■

The term *marking equation* refers to the conclusion

$$\boxed{M_2 = M_1 + C \cdot \mathcal{P}(\tau)}$$

of the lemma.

For example, $\tau = cabda$ is a firing sequence in the net shown in Figure 9. If we want to calculate the marking M reached after this sequence, we may use Lemma 2 as follows:

$$\underbrace{\begin{pmatrix}1\\0\\0\\2\\1\\0\end{pmatrix}}_{M_0} + \underbrace{\begin{pmatrix}1&-1&-1&0&0\\-1&1&1&0&0\\0&0&1&-1&0\\0&0&-1&1&0\\0&0&0&-1&1\\0&0&0&1&-1\end{pmatrix}}_{C} \cdot \underbrace{\begin{pmatrix}2\\1\\1\\1\\0\end{pmatrix}}_{\mathcal{P}(cabda)} = \underbrace{\begin{pmatrix}1\\0\\0\\2\\0\\1\end{pmatrix}}_{M}$$

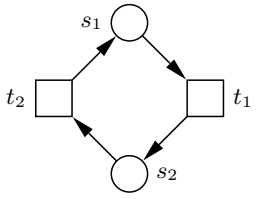
The lemma states that the validity of the marking equation is a *necessary* condition for a given sequence of transitions leading to a certain marking. Stated differently, if the marking equation is not satisfied for a given vector of transition counts, then there is no firing sequence having this vector as a Parikh vector reaching the goal marking.

The marking equation is *not*, in general, a sufficient condition for firability. That is, $M_2 = M_1 + C \cdot y$ and $y \geq 0$ do not necessarily entail $M_1[\tau\rangle M_2$ for some firing sequence τ with $\mathcal{P}(\tau) = y$. As a counterexample, we may consider Figure 10 where the initial marking is empty, i.e. equal to $(0\,0)^\mathsf{T}$.

The above may be expressed in a slightly different way. If $M[\tau\rangle$ and $y = \mathcal{P}(\tau)$, then the two inequalities

$$\boxed{0 \leq y \quad \text{and} \quad 0 \leq M + C \cdot y}$$

are satisfied. This is called the *marking inequality*. A T-vector $y \in \mathbb{N}^T$ is called *realisable* from some marking M if there is a firing sequence $M[\tau\rangle$ with $\mathcal{P}(\tau) = y$.



C	t_1	t_2
s_1	-1	1
s_2	1	-1
$\mathcal{P}(t_1t_2)^\mathsf{T}$	1	1

Fig. 10. A system in which $(0\,0)^\mathsf{T} = (0\,0)^\mathsf{T} + C{\cdot}\mathcal{P}(t_1t_2)$ but $\tau = t_1t_2$ is not firable

If some arbitrary integer-valued vector y satisfies the marking inequality, it is still not guaranteed to be realisable.

2.2 Transposition Lemmata

Apart from the componentwise comparisons of vectors,

$$\begin{aligned} & M \geq M' \iff \forall k: \quad M(k) \geq M'(k) \\ \text{and} \quad & M > M' \iff M \geq M' \wedge M \neq M', \end{aligned}$$

we will also need the following concept of strict comparison.

Definition 14. *Strictly greater*
Let $k \in \mathbb{N}$ and let $M, M' \in \mathbb{Z}^k$ be vectors over $\mathbb{Z}$. Then M is called *strictly greater* than M' (in symbols: $M \gg M'$) if $M(i) > M'(i)$ for all $i \in \{1, \ldots, k\}$. The notion of *strictly less* is defined analogously.

A vector M is called *nonnegative* if $M \geq 0$, *semipositive* if $M > 0$, and *positive* if $M \gg 0$, where 0 denotes the null vector. ■

For example, a Parikh vector is always nonnegative. Some entries may be 0 if the corresponding transition does not occur in the sequence on which the vector is based. However, Parikh vector entries are never negative.

We exploit a useful principle which is sometimes known in Linear Algebra by the name of *transposition principle* or *alternation principle* and which goes back to Gordan [Gor1873], Farkas [Far1902] and others. The following lemma explicates this principle. There are several versions of this lemma, but in the present notes we use only this one.

Lemma 3. *A transposition lemma*
Let A be a matrix with rational entries (that is, entries from the set of rational numbers). Then *exactly one* of the following statements is valid:

(i) There exists a (rational) vector x with $x \gg 0$ and $A^{\mathsf{T}} \cdot x \leq 0$. Here, A^{T} denotes the transposed matrix of A.

(ii) There exists a (rational) vector $y \geq 0$ with $A \cdot y > 0$.

Proof: It is easy to see that (i) and (ii) cannot be true at the same time. This is because (i) $\wedge$ (ii) entails

$$0 \;\geq\; y^{\mathsf{T}} \cdot A^{\mathsf{T}} \cdot x \;>\; 0,$$

which is a contradiction. The first inequality comes from $y \geq 0$ (ii) and from $A^{\mathsf{T}} \cdot x \leq 0$ (i), if the middle product is associated as $y^{\mathsf{T}} \cdot (A^{\mathsf{T}} \cdot x)$. The second inequality comes from $x \gg 0$ (i) and $A \cdot y > 0$ (ii), if the product is associated as $(y^{\mathsf{T}} \cdot A^{\mathsf{T}}) \cdot x$.

The proof that (i) $\vee$ (ii) holds true is non-trivial; the interested reader is referred to [Schr86]. ■

This lemma can easily be lifted to integers for the vectors x and y (whichever exists).

2.3 Structural Boundedness, Infinite Executions, and Dickson's Lemma

This section contains some small examples involving linear-algebraic arguments, including a typical application of the transposition principle. First, we give an elementary linear-algebraic characterisation of structural boundedness. Then, we characterise the existence of infinite firing sequences linear-algebraically.

Proposition 1. *Characterisation of structural boundedness*
Let N be a net and let C be the incidence matrix of N. The following statements are equivalent:
(A) $N = (S, T, F)$ is structurally bounded.
(B) There exists a vector $x \in \mathbb{N}^{|S|}$ with $x \gg 0$ and $C^{\mathsf{T}} \cdot x \leq 0$.

Proof: **(A)⇒(B)** can be shown by contraposition. Assume ¬**(B)**, i.e., there is no vector x as in **(B)**. By Lemma 3, there is some vector $y \in \mathbb{N}^{|T|}$ with $C \cdot y > 0$. We choose some marking M which guarantees that a firing sequence τ with $\mathcal{P}(\tau) = y$ is firable from it, for instance the following:

$$M(s) = \sum_{t \in s^\bullet} (F(s,t) \cdot y(t)), \text{ for } s \in S.$$

Let M' be defined by $M\,[\tau\rangle M'$. The firing lemma yields

$$M' = (\text{ by Lemma 2 }) \; M + C \cdot \mathcal{P}(\tau) = (\text{ by } \mathcal{P}(\tau) = y \text{ }) \; M + C \cdot y > (\text{ by } C \cdot y > 0 \text{ }) \; M.$$

Furthermore, from $M' > M$ we deduce the existence of a place r with $M'(r) > M(r)$. Since τ can be fired arbitrarily often from M' because of $M' > M$, at least the place r is unbounded. Hence ¬**(A)** holds.

To show **(B)⇒(A)**, we choose x such that property **(B)** is satisfied. Let M_1 be an arbitrary marking of N and let $M_1\,[\tau\rangle M_2$ with an arbitrary firing sequence τ. Using **(B)** we get:

$$x^{\mathsf{T}} \cdot M_2 \;=\; x^{\mathsf{T}} \cdot (M_1 + C \cdot p(\tau)) \;=\; x^{\mathsf{T}} \cdot M_1 + x^{\mathsf{T}} \cdot (C \cdot p(\tau)) \;\leq\; x^{\mathsf{T}} \cdot M_1$$

where the first equality follows from the firing lemma and the last inequality from **(B)**. For $s \in S$,

$$x(s) M_2(s) \;\leq\; (\text{ by } x \geq 0 \text{ }) \sum_{r \in S} x(r) M_2(r) \;=\; x^{\mathsf{T}} \cdot M_2 \;\leq\; (\text{ by the above }) \; x^{\mathsf{T}} \cdot M_1.$$

Therefore, $(x^{\mathsf{T}} \cdot M_1)/x(s)$ is an upper bound for the number of tokens on an arbitrary place s in M_2, depending neither on M_2 nor on τ. Therefore, place s is bounded. Since the above is true for arbitrary M_1 and for arbitrary s, Property **(A)** is satisfied. ■

In order to characterise the existence of a marking from which an infinite sequence of transitions can be fired, we exploit a lemma which is useful in several other circumstances as well.

Lemma 4. *Dickson's lemma*

Let $n \in \mathbb{N}$ and let $x_1, x_2, x_3, \ldots$ be an infinite sequence of vectors in $\mathbb{N}^n$. Then there are indices $i_1 < i_2 < i_3 < \ldots$ with

$$x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \ldots$$

Proof: Consider the case that $n = 1$. Then the sequence of vectors is simply a sequence of natural numbers. If one of them occurs infinitely often in the sequence, we are done, since the corresponding subsequence is (weakly) monotonically increasing. Otherwise we choose i_1 as the last occurrence of the minimum, i_2 as the *subsequently* last occurrence of the (new) minimum, and so on.

The case that $n > 1$ can be dealt with by induction and componentwise consideration. From an infinite sequence of vectors with n components, we first choose an infinite subsequence that is (weakly) monotonically increasing with respect to components 1 to $n - 1$. This can be done by induction hypothesis. From this subsequence, we then choose another subsequence which (weakly) increases with respect to the last (nth) component. This can be done as in the case that $n = 1$. The resulting sub-subsequence is (weakly) monotonically increasing with respect to all n components. ■

The lemma can be applied to the sequence of markings occurring in an infinite firing sequence, as follows.

Proposition 2. *Existence of an infinite firing sequence*

For an unmarked net N, there is some marking M_0 such that an infinite firing sequence from M_0 exists, if and only if the system of inequalities $C \cdot y \geq 0,\ y > 0$ has a solution.

Proof: ($\Rightarrow$): Let M_0 be a marking of N with $M_0\,[t_1\rangle M_1\,[t_2\rangle M_2\,[t_3\rangle\ \ldots$ By Lemma 4, there exist indices $i < j$ with $M_i \leq M_j$ and $M_i\,[t_{i+1} \ldots t_j\rangle M_j$. The vector y defined by $y = \mathcal{P}(t_{i+1} \ldots t_j)$ solves the system of inequalities given in the proposition, since: $y \geq 0$, because y is a Parikh vector; $y \neq 0$, because $i < j$; and $C \cdot y \geq 0$ by $M_j = M_i + C \cdot y$ (firing lemma) and by $M_j \geq M_i$.

($\Leftarrow$): Let y be a solution of the system of inequalities given in the lemma. As in the proof of Proposition 1, we can find a 'sufficiently large' marking M that activates a firing sequence σ with $\mathcal{P}(\sigma) = y$. Let M' be the marking defined by $M\,[\sigma\rangle M'$. By the firing lemma, $M' = M + C \cdot y$, whence, by $C \cdot y \geq 0$, we have $M \leq M'$. Hence σ can be iterated indefinitely, and $\sigma\sigma\sigma \ldots$ is firable from M. Moreover, $\sigma\sigma\sigma \ldots$ is an infinite sequence since $\sigma \neq \varepsilon$ because of $y \neq 0$. ■

2.4 S-Invariants and T-Invariants

In the previous section, vectors x (Proposition 1) and y (Proposition 2) satisfied some inequalities, $C^{\mathsf{T}} \cdot x \leq 0$ and $C \cdot y \geq 0$, respectively. The special case that these inequalities become actual equalities, viz. $C^{\mathsf{T}} \cdot x = 0$ and $C \cdot y = 0$, is of particular importance:

Definition 15. *Place invariants and transition invariants*
A vector $x \in \mathbb{Z}^{|S|}$ is called *S-invariant* or *place invariant*, if $C^{\mathsf{T}} \cdot x = 0$.
A semipositive S-invariant x is *minimal* if there is no S-invariant x' with $0 < x' < x$.

A vector $y \in \mathbb{Z}^{|T|}$ is called *T-invariant* or *transition invariant*, if $C \cdot y = 0$. Minimality is defined similarly as for S-invariants. ■

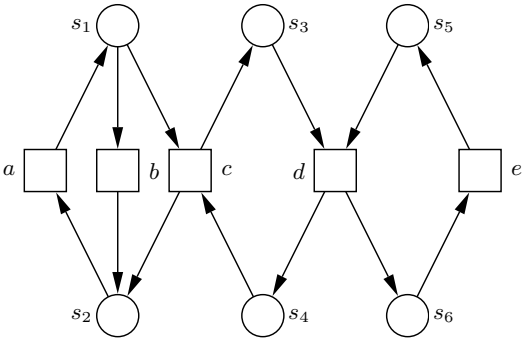
The semipositive, the positive, and the minimal invariants will turn out to be of primary interest.

As an example, consider Figure 11. x_1 is a minimal semipositive S-invariant. x_2 is a non-semipositive S-invariant. x_3 is a semipositive S-invariant which arises from another semipositive S-invariant, namely $(0,0,1,1,0,0)^{\mathsf{T}}$, by multiplication with 2; thus, it is not minimal. x_3 is also the sum of x_1 and x_2. y_1 and y_2 are minimal semipositive T-invariants.

The following lemma follows directly from the firing lemma.

Lemma 5. *Basic properties of S- and T-invariants*
Let x be an S-invariant of N and let M_1, M_2 be markings of N with $M_1\,[\tau\rangle M_2$, for some sequence τ. Then $x^{\mathsf{T}} \cdot M_1 = x^{\mathsf{T}} \cdot M_2$.
Let M be a marking of N with $M\,[\tau\rangle M$ for some sequence τ. Then $\mathcal{P}(\tau)$ is a T-invariant of N.
Conversely, if $M\,[\tau\rangle$ and $\mathcal{P}(\tau)$ is a T-invariant of N, then $M\,[\tau\rangle M$. ■

Informally, the first part of this lemma states that the x-weighted marking on any S-invariant x is constant. In particular, if a net has a positive S-invariant, then it is necessarily structurally bounded. The second part of the lemma states that any reproduction sequence generates a T-invariant. In particular, any reproduction sequence containing every transition at least once, generates a positive T-invariant. A weak converse also holds true: if $\mathcal{P}(\tau)$ is a T-invariant, then $M\,[\tau\rangle M$ for any marking M enabling τ.



C	a	b	c	d	e	x_1	x_2	x_3
1	1	−1	−1	0	0	1	−1	0
2	−1	1	1	0	0	1	−1	0
3	0	0	1	−1	0	0	2	2
4	0	0	−1	1	0	0	2	2
5	0	0	0	−1	1	0	0	0
6	0	0	0	1	−1	0	0	0
y_1^{T}	1	1	0	0	0			
y_2^{T}	1	0	1	1	1			

Fig. 11. The example of Figure 8, with some S- and T-invariants

2.5 Positive S-Invariants and T-Invariants

A positive S-invariant, that is, one which assigns a number ≥ 1 to every place, is said to *cover* the net. Similarly, a positive T-invariant is said to *cover* a net. As these properties occur frequently, often in connection with well-formedness, we abbreviate them as follows:

(PS): The net under consideration is covered by a positive S-invariant.
(PT): The net under consideration is covered by a positive T-invariant.
(WF): The net under consideration is well-formed, i.e., it has a live and bounded marking.

(WF) is stronger than (PT), because of the following lemma. However, (WF) is not stronger than (PS), since there are Petri nets satisfying (WF) but not (PS). Finding an example is left as an exercise to the reader. (This is not entirely trivial.)

Proposition 3. *On the existence of positive T-invariants*
Let N be a well-formed Petri net. Then N has a positive T-invariant.

Proof: Let M be a live and bounded marking of N. By liveness, there exists an infinite firing sequence $\tau = \tau_1\tau_2\tau_3\ldots$ such that every sequence τ_i contains *all* transitions of N. Define markings M_i by $M[\tau_1\ldots\tau_i\rangle M_i$. By boundedness, not all markings M_i can be different. Hence there are two indices k, j with $k < j$ and $M_k = M_j$. The subsequence $M_k\,[\tau_{k+1}\ldots\tau_j\rangle M_j$ is repetitive, as it reproduces the marking $M_k{=}M_j$. Thus, $\mathcal{P}(\tau_{k+1}\ldots\tau_j)$ is a T-invariant by the second part of Lemma 5, which is positive since $\tau_{k+1}\ldots\tau_j$ has the nonempty suffix τ_j and thus contains at all transitions, by definition of τ_j. ■
If (PS) and (PT) are true for some net, then there are repercussions on its graph-theoretical structure.

Proposition 4. *Cycle-coveredness of nets covered by positive S- and T-invariants*

Let N be a Petri net satisfying (PS) and (PT). Then N is covered by cycles.

Proof: (*Sketch.*) Let $N = (S, T, F)$ and choose x such that $C^{\mathsf{T}}{\cdot}x = 0$ and $x \gg 0$, and y such that $C{\cdot}y = 0$ and $y \gg 0$. Consider an arrow (u, v) in F. We want to prove that there is a directed path from v to u.

First Case: $u \in S$ and $v \in T$.
The basic proof idea is to restrict y to transitions 'after' v. Let $y'\colon T \to \mathbb{N}$ be defined as follows:

$y'(t) = y(t)$ if a directed (possibly empty) path leads from v to t in N,
$y'(t) = 0$ for all other transitions t.

It is then possible to show that (i) y' is a T-invariant, and (ii) some input transition $w \in {}^\bullet u$ satisfies $y'(w) > 0$. By the definition of y', a directed path leads in N from v to w, and therefore also from v to u.

Second Case: $u \in T$ and $v \in S$.
Consider the *dual net* $N^d = (T, S, F)$, in which places and transitions are exchanged but arrows are retained. The incidence matrix of N^d is $-C^{\mathsf{T}}$. Hence x is a positive T-invariant and y is a positive S-invariant of N^d, and the second case is reducible to the first case. ■

Corollary 2. *Strong connectedness of nets covered by positive S- and T-invariants*

Let N be a weakly connected Petri net containing a positive S-invariant and a positive T-invariant.
Then N is strongly connected. ■

2.6 Rank, Conflict Clusters, and Sets of Presets

There are some interesting connections between the structural liveness of a Petri net and the rank of its incidence matrix C. The *column rank* (*row rank*) of C is defined as the maximal number of linearly independent column (row, respectively) vectors in C. Since, as is known from your favourite course on Linear Algebra, the column rank and the row rank of any matrix C are identical, the *rank* of C is simply defined as one of them, say the column rank. The rank of a Petri net N is defined as the rank of its incidence matrix.

In the remaining part of this section, we will assume that N is weakly connected and *plain*, that is, the function F does not yield values greater than 1 (or, equivalently, the matrices $\mathbb{B}$ and $\mathbb{F}$ have values in the set $\{0, 1\}$).

A first observation is that the rank of C is less than $|T|$, the number of transitions, provided that N is covered by a positive T-invariant. This is simply because $C \cdot y = 0$ and $y \gg 0$ just means that some positive linear combination of the columns of C equals 0, which means that its columns are linearly dependent and the rank of C cannot exceed $|T| - 1$. We may combine this observation with Proposition 3, to see that any well-formed net has column rank $\leq |T| - 1$.

As a special case, consider a simple directed cycle and a function which assigns the number 1 to every transition of the cycle, as shown on the left-hand side of Figure 12. This is already a positive T-invariant, and the column rank of the

Fig. 12. A simple cycle (l.h.s.) and a modification (r.h.s.)

net actually equals $|T|-1$. Suppose now that we change such a cycle slightly by putting two successive transitions into conflict rather than sequence, as depicted on the right-hand side of Figure 12. Then, in some reproducing firing sequence, either one or the other can be chosen. In this way, we get two semi-positive, non-positive T-invariants. Their sum is a positive T-invariant covering the net with column rank $|T|-2$. There is one place less in the net (compared to the left hand side of Fig. 12), leading to less potential conflicts between transitions. Thus, one might be led to suspect a connection between the rank of a net and its 'degree of conflict'. The notions of a *conflict cluster* and of a *preset*, as follows, are designed to make this suspicion more precise. Informally, both play a role in capturing the 'degree of conflict' of a net.

Definition 16. *Conflict clusters, and the set of presets*
Let $N = (S, T, F)$ be a plain Petri net.

Fot $t, t' \in T$, let $t \sim_0 t'$ if ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ (i.e., if there is a potential conflict between t and t'). Let $\sim \subseteq T \times T$ be the reflexive and transitive closure of $\sim_0$. A *conflict cluster* of N is defined as an equivalence class of the equivalence relation generated by $\sim$. The set of all conflict clusters of N is denoted by CC_N.

The set of all non-empty *presets* of N is defined as $PRESETS_N = \{{}^\bullet t \mid t \in T \wedge {}^\bullet t \neq \emptyset\}$. ■

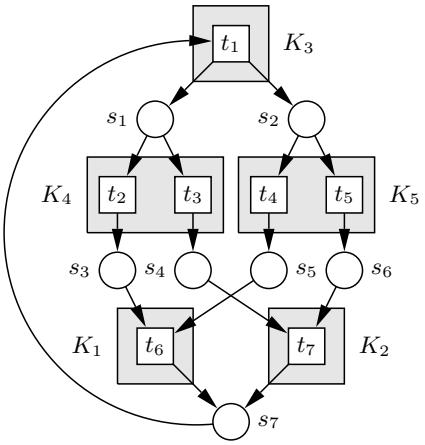
Notice that in the simple cycle on the left-hand side of Figure 12, the relation $\sim_0$ is the identity relation, and we have three conflict clusters. Also, there are three presets. The net has rank $2 = |T|-1$ overall. On the right-hand side of Figure 12, however, $\sim_0$ is not the identity since we have $t_2 \sim_0 t_3$. In all, there are two conflict clusters, as well as two presets. The rank of the net's matrix is $1 = |T|-2$.

2.7 Sufficient and Necessary Conditions for Structural Liveness

Theorem 2. *Sufficient condition for the existence of a live marking*
Assume that N is a weakly connected, plain net covered by a positive S-invariant and a positive T-invariant.
If the rank of C is strictly less than $|CC_N|$, then there exists a live marking of N.

Proof: (*Sketch.*)
The proof may be done by contraposition. Supposing that no live marking of N exists, the column rank of C is shown to be $\geq |CC_N|$. The proof proceeds in several steps, starting with a suitably chosen non-live marking M_1 and constructing exactly $|CC_N|$ linearly independent column vectors contained in C.

First, we consider an initial marking M_1 such that all places of all conflict clusters are marked with some token. By assumption, M_1 is not live. Using this, and also the strong connectedness obtained by Corollary 2, it may be shown that a firing sequence $M_1[\tau\rangle M_2$ exists, such that in M_2, every conflict cluster contains a transition with an unmarked input place. (This argument is invalid if there are arc weights greater than 1.) Linearly independent entries in C can be then constructed as follows. The sequence τ is scanned backwards, such that for



C	t_1	t_2	t_3	t_4	t_5	t_6	t_7
s_1	1	−1	−1	0	0	0	0
s_2	1	0	0	−1	−1	0	0
s_3	0	1	0	0	0	−1	0
s_4	0	0	1	0	0	0	−1
s_5	0	0	0	1	0	−1	0
s_6	0	0	0	0	1	0	−1
s_7	−1	0	0	0	0	1	1

Fig. 13. A non-structurally-live net, its conflict clusters (l.h.s.), and its incidence matrix (r.h.s.)

every conflict cluster, the *last* transition in τ is recorded. It can be shown that the corresponding entries in C are linearly independent, and since τ contains at least one transition from every conflict cluster, the number of transitions so obtained equals $|CC_N|$. ■

Consider, for example, the Petri net shown on the left-hand side of Figure 13 and its incidence matrix, shown on the right-hand side. The net is plain, and it satisfies both (PS) and (PT), the verification of which is left to the reader. It has 5 conflict clusters, also shown on the left-hand side of the figure. There exists no live marking for this net. The theorem claims that the rank of C should be ≥ 5, and indeed, it actually equals 5. For instance, the first five columns are linearly independent, while the remaining two columns can be linearly combined from them.

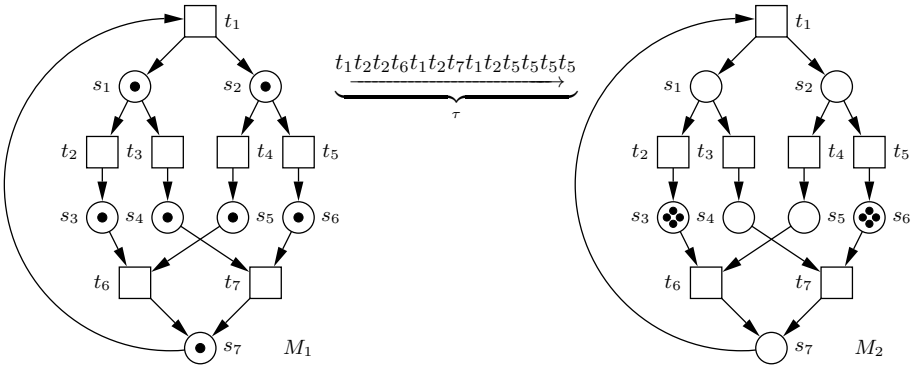
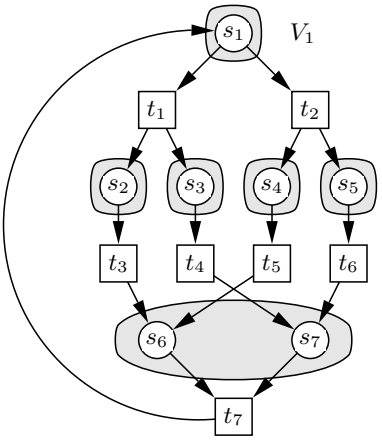


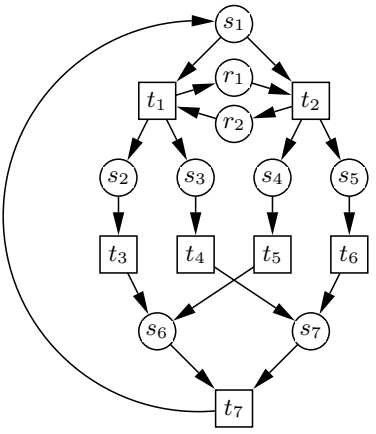
Fig. 14. $M_1\,[\tau\rangle M_2$, and in M_2, every conflict cluster contains a transition with an unmarked preplace



C	t_1	t_2	t_3	t_4	t_5	t_6	t_7
s_1	-1	-1	0	0	0	0	1
s_2	1	0	-1	0	0	0	0
s_3	1	0	0	-1	0	0	0
s_4	0	1	0	0	-1	0	0
s_5	0	1	0	0	0	-1	0
s_6	0	0	1	0	1	0	-1
s_7	0	0	0	1	0	1	-1

Fig. 15. A structurally live net, its set of presets (l.h.s.), and its incidence matrix (r.h.s.)

To trace the constructions in the proof, we may consider a marking M_1 putting exactly one token on each place, such as shown on the left-hand side of Figure 14. This guarantees that every place in every conflict cluster has a token. A marking M_2, reachable from M_1, such that every transition in every conflict cluster has at least one unmarked input place is shown on the right-hand side of Figure 14, and a firing sequence τ with $M_1\,[\tau\rangle M_2$ is also shown. In the final step of the proof, the following transitions are recorded, in this order: t_5 (for K_5), then t_2 (for K_4), and then, similarly, t_1, t_7, t_6. The corresponding entries in C are linearly independent.



C	t_1	t_2	t_3	t_4	t_5	t_6	t_7
s_1	-1	-1	0	0	0	0	1
s_2	1	0	-1	0	0	0	0
s_3	1	0	0	-1	0	0	0
s_4	0	1	0	0	-1	0	0
s_5	0	1	0	0	0	-1	0
s_6	0	0	1	0	1	0	-1
s_7	0	0	0	1	0	1	-1
r_1	1	-1	0	0	0	0	0
r_2	-1	1	0	0	0	0	0

Fig. 16. The net of Figure 15 with a regulation circuit $\{t_1, r_1, t_2, r_2\}$ (l.h.s.), and its incidence matrix (r.h.s.)

Theorem 3. *Necessary condition for the existence of a live marking*
Assume that N is a weakly connected, plain net covered by a positive S-invariant. If there exists a live marking of N, then the rank of C is strictly less than $|PRESETS_N|$.

Proof: (*Sketch.*)
Let N be a net which has a live marking and is covered by a positive S-invariant. By Proposition 3, there exists a positive T-invariant. Thus, N satisfies (PT).

If the number $m = |PRESETS_N|$ is 0, then no transition has any input place. Since there are at least one transition and one place and the net is weakly connected, there is some transition without any input place but with some output place. Such a net cannot satisfy (PS). Hence $1 \leq m \leq |T|$, where $m = |T|$ in case no two transitions have a common preset. The theorem can be proved by induction on $|T| - m \geq 0$.

Base: Suppose $m = |T|$. Because N satisfies (PT), the rank of C is less than $|T| = |PRESETS_N|$.

Step: Suppose $m < |T|$. Then there exist at least two transitions with the same preset. Let U with $|U| \geq 2$ be some set of transitions all of which have the same preset. (For instance, consider $U = \{t_1, t_2\}$ on the left-hand side of Figure 15.) Define $N[U]$ as N, augmented with a *regulation circuit* through the transitions of U (such as shown on the left-hand side of Figure 16, cf. r_1 and r_2). The following can be observed:

1) $N[U]$ satisfies (PS) and is structurally live. To show structural liveness of $N[U]$, a live marking of N can be augmented by sufficiently many tokens on the places of the regulation circuit; an upper limit for the number of such tokens can be derived from (PS).
2) $|PRESETS_{N[U]}| = |PRESETS_N| + |U| - 1 > |PRESETS_N|$ by properties of $N[U]$ and by $|U| \geq 2$.

Because of the inequality in 2), the induction hypothesis can be applied to $N[U]$, entailing

$$(\text{rank of } N[U]) \quad \leq \quad |PRESETS_N| + |U| - 2. \tag{3}$$

It can moreover be shown that (rank of N) $+ |U| - 1 \leq$ (rank of $N[U]$), which can be combined with (3), yielding (rank of N) $\leq |PRESETS_N| - 1$ and ending the inductive proof. ∎

As an example, see Figure 15 which shows on its left-hand side the dual of the previous example. This net is also plain and satisfies (PS). It is structurally live as well; a live and bounded (even safe) marking of it has already been shown in Figure 7. The theorem claims that the number of presets should be larger than the rank of the incidence matrix. Indeed, the number of presets is 6, while the rank of the incidence matrix is 5, just as before.

To trace the inductive proof, we may consider the set $U = \{t_1, t_2\}$ in Figure 15. Then $N[U]$ is shown on the left-hand side of Figure 16, and its incidence

matrix on the right-hand side of Figure 16 has rank 6. The inequalities claimed in the proof can thus be verified.

To see that weak connectedness is required as a precondition of this theorem, consider the net consisting of an isolated place, an isolated transition, and any marking. Such a net has a positive S-invariant and is live, but its rank is 0 and the number of presets is also 0, so the inequality claimed in the theorem fails to hold.

2.8 Bibliographical Remarks and Further Reading

The use of the incidence matrix, of S- and T-invariants, and of Dickson's and Farkas' lemma date back to early work in [KM69, CHEP71, GL73] and also to work by Kurt Lautenbach [Lau73]. Often in the literature, 'Dickson's lemma' denotes a statement which may be more general or slightly different from the one we used. Innocent as it might seem, Dickson's lemma can also be viewed as a (very restricted) special case of one of the most famous new results in graph theory, the *graph minor theorem*, cf. [Die10].

The connections between the rank of the incidence matrix and structural liveness were discovered for free-choice nets – to be defined in the next section – by a group around Manuel Silva in Zaragoza [CCS91]. In the context of free-choice Petri nets, these results are also contained – with improved proofs – in the textbook by Jörg Desel and Javier Esparza [DE95]. In this section, we have presented them independently of the free-choice property, and this presentation is due to Jörg Desel [Des92, Des98].

3 Graph-Theoretical Structure of Petri Nets

Graph-theoretically speaking, a Petri net is a *bipartite directed multigraph*. The term 'bipartite' refers to the fact that the set of nodes is divided into two disjoint sets, places and transitions, such that arcs connect nodes from one set with nodes of the other set, but never two nodes of the same set. The term 'multigraph' refers to the possible non-plainness of a Petri net, in the sense that there may be several arcs in the same direction between two nodes. In the remainder of this tutorial, we will neglect such multiplicities, however:

> Henceforth, all Petri nets considered in definitions or in results, will be assumed to be *plain*.

In the language of graph theory, this means that we consider *bipartite digraphs*. This section is devoted to exhibiting a few results by which the graph-theoretical structure of such a bipartite digraph (called a Petri net N) can be related to properties of the reachability graph of N.

In Petri net literature, one finds various constructions for turning non-plain Petri nets into 'behaviourally equivalent' plain ones. These constructions usually involve new places, new transitions and new tokens. In each individual case, however, it has to be checked whether the properties one is interested in are

stable with respect to such transformations, or whether definitions and results one wishes to apply can be transferred easily from the plain to the non-plain case. For the definitions and results described in this and in the next section, such considerations can be non-trivial.

3.1 Some Simple Observations

Let us start our considerations with isolated places (i.e., places s such that ${}^\bullet s = \emptyset = s^\bullet$) and isolated transitions (i.e., transitions t such that ${}^\bullet t = \emptyset = t^\bullet$). An isolated place neither loses any of the tokens it has initially, nor does it gain any new tokens. Therefore, it impacts neither on liveness nor on boundedness properties, and we might as well (and will) exclude such places. An isolated transition can occur indefinitely often, neither needing any input tokens nor producing any output tokens. As such, it also has no effect on (strong or weak) liveness or on boundedness and we will exclude isolated transitions from consideration as well:

> Henceforth, all Petri nets considered will have no isolated places and no isolated transitions.

Next, we consider places and transitions with mixed empty and non-empty pre- or postsets. A place s with ${}^\bullet s \neq \emptyset = s^\bullet$ destroys either liveness or boundedness, because if the transitions in ${}^\bullet s$ are live, then s is most certainly not bounded. A place s with ${}^\bullet s = \emptyset \neq s^\bullet$ destroys liveness, because the transitions in $s^\bullet$ can fire at most as many times as there are tokens on s initially. A transition t with ${}^\bullet t \neq \emptyset = t^\bullet$ destroys either liveness or boundedness, because it is not live, unless unboundedly many tokens can be assembled on the places in ${}^\bullet t$. A transition t with ${}^\bullet t = \emptyset \neq t^\bullet$ destroys boundedness, because it can fire indefinitely often in isolation, putting unboundedly many tokens on every place in $t^\bullet$.

These observations can, in fact, be extended to the following, which is a counterpart of Proposition 4:

Proposition 5. *Cycle-coveredness of well-formed nets*
Let N be a well-formed plain Petri net. Then N is covered by cycles.

Proof: (*Sketch.*)
Let (u, v) be some arc in N.

If $u \in T$, then $v \in S$. If there was no path from v to u, then liveness would allow the part of the net which does not depend on v to fire sufficiently many times in order to put arbitrarily many tokens on v, contradicting boundedness.

If $u \in S$, then $v \in T$. If there was no path from v to u, then the liveness of v could only be guaranteed if arbitrarily many tokens could be assembled on u, again prompting a contradiction. ■

Corollary 3. *Strong connectedness of well-formed nets*
Let N be a weakly connected, well-formed plain Petri net. Then N is strongly connected. ■

Weak connectedness is not, usually, a strong requirement. For example, if a net is not weakly connected, one may analyse its weakly connected components separately. Sometimes, however, it is useful to consider non-weakly-connected substructures of an otherwise, weakly or strongly, connected net.

3.2 S-Nets, T-Nets, and Free-Choice Nets

Using the graph-theoretical structure of a Petri net, it is possible to define restrictive, but meaningful *Petri net classes*. Such classes can be useful in practice, but also in theory, since problems which are hard in general can sometimes be made more tractable by studying them in one of the restricted net classes first. We shall analyse problems such as liveness and boundedness for a range of structurally restricted classes of Petri nets.

S-nets forbid synchronisation and 'splitting' as shown on the left-hand side of Figure 17. T-nets prevent 'merging' and conflicts as shown on the right-hand side of Figure 17.

Definition 17. *S-nets and T-nets*
A plain net $N = (S, T, F)$ is called an *S-net* if $\forall t \in T\colon |{}^\bullet t| \leq 1 \geq |t^\bullet|$.
A plain marked net $N = (S, T, F, M_0)$ is an *S-system* if (S, T, F) is an S-net.
A plain net $N = (S, T, F)$ is called a *T-net* if $\forall s \in S\colon |{}^\bullet s| \leq 1 \geq |s^\bullet|$.
A plain marked net $N = (S, T, F, M_0)$ is a *T-system* if (S, T, F) is a T-net. ■

T-systems satisfy a basic token conservation property. For a marking M and a place set $S' \subseteq S$, let

$$M(S') \;=\; \sum_{s \in S'} M(s),$$

and for a cycle (that is, a simple, closed path) γ, let

$$M(\gamma) = M(S'), \quad \text{where } S' \text{ is the set of places on } \gamma.$$

We say that S' is *token-empty* (*token-free*) or *marked*, depending on whether $M(S') = 0$ or $M(S') > 0$.

Lemma 6. *Elementary property of T-systems*
Let $N = (S, T, F, M_0)$ be a T-system and let $M \in [M_0\rangle$. For every cycle γ of (S, T, F), $M(\gamma) = M_0(\gamma)$.

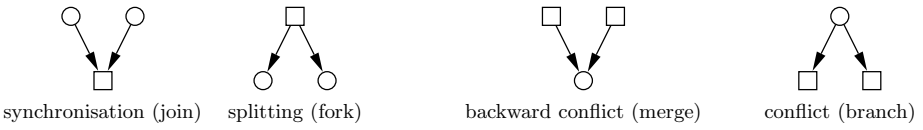


Fig. 17. Forbidden structures: S-nets (l.h.s.) and T-nets (r.h.s.)

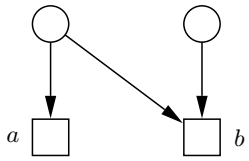


Fig. 18. Forbidden structure in FC-nets

Proof: Consider a cycle γ and the effect of firing a transition t. If t lies on γ, firing t moves exactly one token on γ. If t does not lie on γ, firing t does not affect the tokens on γ. ■

Note that whilst Definition 17 could be applied *verbatim* to non-plain nets, this is not necessarily meaningful. In particular, Lemma 6 would no longer be true.

Next, we will define a class of nets that encompasses both S-nets and T-nets called *free-choice nets* (or *FC-nets*, for short). FC-nets can be viewed as a 'smallest common generalisation' of S-nets and T-nets. In FC-nets, all structures shown in Figure 17 are allowed, but a combination of two of them, such as shown in Figure 18, is disallowed.

Definition 18. *Free-choice nets (FC-nets)*
A plain net $N = (S, T, F)$ is called an *FC-net* if

$$\forall t_1, t_2 \in T\colon {}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset \Rightarrow {}^\bullet t_1 = {}^\bullet t_2. \tag{4}$$

A plain marked net $N = (S, T, F, M_0)$ is an *FC-system* if (S, T, F) is an FC-net. ■

The free-choice property is not satisfied in Figure 18, since ${}^\bullet a \cap {}^\bullet b \neq \emptyset$ and ${}^\bullet a \neq {}^\bullet b$. Originally, the class of free-choice nets was defined more restrictively. A (plain) net (S, T, F) was called free-choice if

$$\forall t_1, t_2 \in T\colon {}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset \;\Rightarrow\; |{}^\bullet t_1| = |{}^\bullet t_2| = 1. \tag{5}$$

The class of nets defined in 18 was originally called *extended free-choice*. Since most important properties and results either hold for both classes or are easily transferred from one to the other, we feel justified in ignoring this distinction for the time being. When we explicitly refer to the class defined by (5) (and this will occur only once, in section 3.7), then we speak of fc-nets rather than FC-nets.

Every free-choice net satisfies the following property which is symmetric to its defining property (4):

$$s_1{}^\bullet \cap s_2{}^\bullet \neq \emptyset \;\Rightarrow\; s_1{}^\bullet = s_2{}^\bullet \tag{6}$$

In fact, (6) is equivalent to (4) and could have been used as an alternative definition of the FC-property.

The nomenclature 'free choice' can be explained in the following way. Suppose that in a free-choice net, some marking M activates a transition t. By (4), all

transitions in the conflict cluster $({}^\bullet t)^\bullet$ of t are activated, and one may freely choose between firing any of them.

Another interesting property of a free-choice net N is that, unless there are transitions t with ${}^\bullet t = \emptyset$, its conflict clusters CC_N and its presets $PRESETS_N$ are in 1-1 correspondence with each other. For a conflict cluster $U \in CC_N$, the set ${}^\bullet U$ is well-defined because any two transitions in U have the same presets by (4), and if it is nonempty, then it is a preset in $PRESETS_N$. Conversely, for any preset $R \in PRESETS_N$, the set $R^\bullet$ is well-defined by (6), and it is a conflict cluster. Moreover, $U = ({}^\bullet U)^\bullet$ and $R = {}^\bullet(R^\bullet)$ for clusters U with ${}^\bullet U \neq \emptyset$ and for presets R, which means that the correspondence is indeed one-to-one, unless there are transitions with empty presets.

If (PS) holds for an FC-net, or if it is well-formed, transitions t with ${}^\bullet t = \emptyset$ are absent, and then the pleasant property $|CC_N| = |PRESETS_N|$ is valid. Theorems 2 and 3 can therefore be combined for FC-nets as follows:

Corollary 4. *Characterisation of the existence of a live marking in FC-nets*
Let N be a weakly connected plain FC-net satisfying (PS) and (PT).
N has a live marking if and only if its rank is strictly less than $|CC_N|$. ■

All S-nets and all T-nets are free-choice nets. Therefore, the last corollary also applies to such nets. The reader is encouraged to verify and simplify it separately for S-nets and for T-nets. However, the class of FC-nets is larger than the union of the classes of S-nets and T-nets. In the previous sections, several FC-nets which are neither S-nets nor T-nets were exhibited, such as, for example, in Figures 13 and 15.

3.3 A Liveness Criterion for FC-Systems

Corollary 4 gives a structural criterion for the *existence* of a live marking in an unmarked FC-net. Next, we characterise the circumstances under which a *given* marking is live in an FC-system. The characterisation uses two graph-theoretical structures that can meaningfully be defined for any Petri net, siphons and traps.

Definition 19. *Siphons and traps*
Let $N = (S, T, F)$ be a plain Petri net. A set $D \subseteq S$ is called *siphon* or *d-set* if ${}^\bullet D \subseteq D^\bullet$.
A set $Q \subseteq S$ is called *trap* or *t-set* if $Q^\bullet \subseteq {}^\bullet Q$. ■

According to this definition, the empty set $\emptyset \subseteq S$ is both a siphon and a trap. Moreover, it is not difficult to see that the union of two siphons (traps) is also a siphon (a trap, respectively). This property is, however, not valid for the intersection.

Lemma 7. *Elementary properties of siphons and traps*
Let D be a siphon, let $M(D) = 0$ and let $M' \in [M\rangle$. Then $M'(D) = 0$.
Let Q be a trap, let $M(Q) > 0$ and let $M' \in [M\rangle$. Then $M'(Q) > 0$.

Proof: Assume $M[t\rangle M'$ with $M(D)=0$ and $M'(D)>0$. Then necessarily $t \in {}^\bullet D$. By ${}^\bullet D \subseteq D^\bullet$, also $t \in D^\bullet$, contradicting $M[t\rangle \wedge M(D)=0$. Thus if D is token-empty, D remains token-empty.

Assume $M[t\rangle M'$ with $M(Q)>0$ and $M'(Q)=0$. Then necessarily $t \in Q^\bullet$. By $Q^\bullet \subseteq {}^\bullet Q$, also $t \in {}^\bullet Q$, contradicting $M[t\rangle \wedge M'(Q)=0$. Thus once Q is marked, Q remains marked. ■

This lemma can be applied in a special circumstance. Consider a net with initial marking M_0 which has some siphon D, and inside D, some trap Q with $M_0(Q)>0$. By Lemma 7, such a siphon D can never be completely emptied of tokens. It turns out that for FC-nets, liveness is already guaranteed if this condition holds for *every* siphon $D \neq \emptyset$:

Theorem 4. *The Commoner/Hack Criterion* CHC

Let $N = (S, T, F, M_0)$ be a free-choice system. The following two properties are equivalent:

(i) For all siphons $D \subseteq S$ with $D \neq \emptyset$ there is a trap $Q \subseteq D$ such that $M_0(Q) > 0$ } CHC

(ii) N is live

Proof: (*Sketch.*)
(i)⇒(ii) can be proved by contraposition. Suppose that M_0 is not live. Then there are $t \in T$ and $M \in [M_0\rangle$ such that t is dead at M. By the FC property, it can easily be shown that there is a place $s \in {}^\bullet t$ which is token-empty at all markings reachable from M. Then every transition in ${}^\bullet s$ is also dead at M. By a backtracking (i.e., repeating this argument), a siphon which is token-empty at M can be constructed. The siphon constructed by this algorithm cannot contain a trap which is marked at M_0, because such a trap could not have been emptied of tokens completely. That is, CHC fails to hold.

(ii)⇒(i) can be proved by contradiction. Suppose that $N = (S, T, F, M_0)$ is an FC-system which does *not* satisfy the Commoner/Hack Criterion CHC and, at the same time, is live. We deduce a contradiction.

Because of $\neg$ CHC, there exists a siphon $D \neq \emptyset$ which does not contain a trap marked at M_0. In particular, the (set-theoretically) largest trap Q in D is unmarked at M_0; note that Q always exists because $\emptyset$ is a trap, and that it is unique because the union of two traps is again a trap. It may be the case that $q = \emptyset$.

For the contradiction, we wish to show that this particular siphon D can be made completely free of tokens, because it is then that all of its output transitions (and there is at least one, due to the absence of isolated places) are dead, contradicting liveness. In this respect, the set $D \setminus Q$ is of critical importance, because it could contain tokens and it might be possible to move some of them onto Q. Once Q has a token, the chance of obtaining token-emptiness of D is obliterated. We need to show that starting from M_0, we can find some firing sequence which removes all tokens from $D \setminus Q$ without, at the same time, putting any tokens on Q. This is done by means of *allocations*.

An allocation is essentially a conflict resolution rule, picking exactly one transition out of a conflict cluster. If the transitions of the cluster are all enabled simultaneously, firing *according to an allocation* means that the allocated transition will be chosen, rather than any other. Now for the proof, it can be shown that there exists an allocation α which keeps removing tokens from $D \setminus Q$ without continually putting tokens back there, and also not onto Q. Firing according to α will eventually make $D \setminus Q$ token-free while keeping Q token-free. Eventually, both Q and $D \setminus Q$ are empty, and a token-free nonempty siphon with at least one output transition is obtained, contradicting the liveness of N. ■

Note that Condition CHC mentions only the initial marking and the two graph-theoretical structures of trap and siphon. In particular, it does not refer to the reachability set $[M_0\rangle$ or to the reachability graph of N. When property CHC is tested algorithmically, it suffices to consider only the minimal siphons and in each of them, the maximal trap. Still, in the worst case there may be exponentially many minimal siphons.

The next examples demonstrate that the premise of free-choiceness cannot be omitted in any of the two directions of the liveness theorem. The left-hand side of Figure 19 presents a non-FC-system satisfying condition CHC but failing to be live. Note, that this system is deadlock-free, though, a property that holds for all systems satisfying CHC in general, even if they are non-FC-systems. The right-hand side of Figure 19 shows a non-FC-system which is live but fails to satisfy condition CHC. To see this, note that $\{s_1, s_2, s_3, s_4\}$ is a siphon which does not contain any marked trap. The free-choice property is violated at place s_4 and its output transitions.

The reader is invited to check what becomes of CHC in the special cases of S-systems. For T-systems, one has the following result:

Theorem 5. *Liveness and realisability of Parikh vectors in T-systems*
Let $N = (S, T, F, M_0)$ be a plain T-system. The following are equivalent:

a) N is live;
b) all places $s \in S$ satisfy ${}^\bullet s \neq \emptyset$ and all (elementary) cycles carry at least one token under M_0;

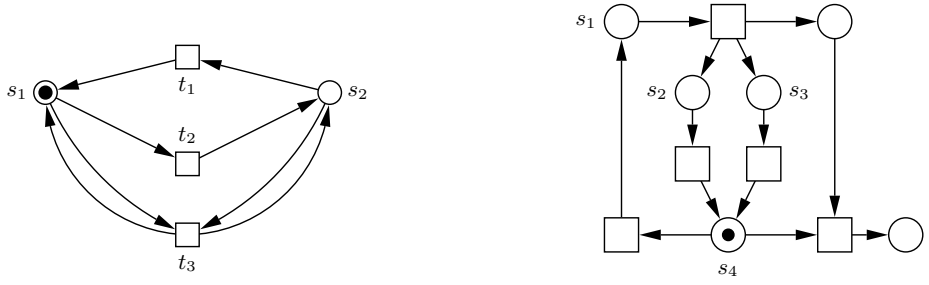


Fig. 19. Non-free-choice counterexamples to Theorem 4(⇒) and 4(⇐), respectively

c) all places $s \in S$ satisfy ${}^\bullet s \neq \emptyset$ and the Parikh vector 1 is *realisable*, that is, there is some firing sequence τ such that $M_0\,[\tau\rangle M$ and every transition occurs exactly once in τ.

Proof: (*Sketch.*)
In a T-system, minimal siphons are either singletons $\{s\}$ for a place $s \in S$ with ${}^\bullet s = \emptyset$, or elementary cycles. The former cannot contain any trap because of the absence of isolated places, and the maximal trap in an elementary cycle is the cycle itself. Thus condition b) is exactly what CHC reduces to for T-systems, and the equivalence between a) and b) turns out to be the counterpart of Theorem 4 for T-systems.

c)⇒b): If 1 is realisable, there can be no token-free cycles.

a)⇒c): If a place $s \in S$ satisfies ${}^\bullet s = \emptyset$, consider $t \in s^\bullet$. This transition exists due to the absence of isolated places. Then t can fire at most $M_0(s)$ times, i.e., there is some reachable marking at which t is dead.

The firability of 1 can be shown by induction on the number of transitions. If $T = \{t\}$, liveness implies that t can be fired once from M_0. Suppose $|T| > 1$ and $t \in T$ such that $M_0[t\rangle$. Then N can be transformed into another live T-system N' by erasing t and 'merging' input places and output places of t in an appropriate way. By induction hypothesis, a suitable firing sequence τ' exists in N'. Then $\tau = t\tau'$ is a suitable firing sequence in N. ■

3.4 A Boundedness Criterion, and Some Coverability Results, for Live FC-Systems

In the previous section, an exact liveness criterion for FC-systems was described. In this section, this discussion is extended by presenting an exact characterisation of the boundedness of a live FC-net. We still assume all nets to be plain, and more graph-theoretical concepts are needed. In particular, we define two particular kinds of subnets.

Definition 20. *S-components and T-components*
Let $N = (S, T, F)$ be a plain net and let N_1 be the subnet $N(S_1, T_1)$ for some $S_1 \subseteq S$ and $T_1 \subseteq T$.

N_1 is called an *S-component* of N if $T_1 = {}^\bullet S_1 \cup S_1^\bullet$ (taking the preset and the postset in N) and $\forall t \in T_1 : |{}^\bullet t \cap S_1| \leq 1 \geq |t^\bullet \cap S_1|$.

N_1 is called a *T-component* of N if $S_1 = {}^\bullet T_1 \cup T_1^\bullet$ and $\forall s \in S_1 : |{}^\bullet s \cap T_1| \leq 1 \geq |s^\bullet \cap T_1|$.

N_1 is called *strongly connected* (inside N) if N_1 is strongly connected (as a separate net). ■

On the left-hand side of Figure 20, a net N is shown. The right-hand side of the figure shows three of its subnets, N_1, N_1' and N_1''. N_1 is a (non-strongly-connected) T-component but not an S-component. N_1' is a strongly connected S-component, but not a T-component, since the property $S_1 = {}^\bullet T_1 \cup T_1^\bullet$, is

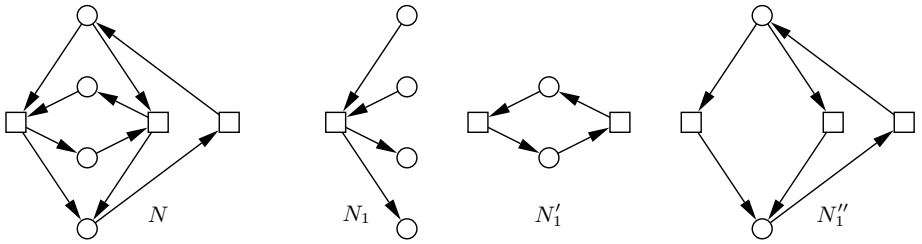


Fig. 20. Sample S- and T-components

violated in N. N_1'' is another strongly connected S-component. In N there are no strongly connected T-components.

Let N_1, with place set S_1, be a strongly connected S-component of (S, T, F). It is easy to verify that the S-vector having a 1 at places in S_1 and a 0 at places in $S \backslash S_1$ is an S-invariant. Similarly, every strongly connected T-component defines a binary T-invariant with entries in $\{0, 1\}$. In the following, unless specified otherwise, we consider only strongly connected S- and T-components.

Theorem 6. *Covering by S-components, and a boundedness criterion for live FC-systems*

Let $N = (S, T, F, M_0)$ be a (plain) live FC-system and let $s \in S$.

(1) A place s is m-bounded ($m \in \mathbb{N}$, $m \geq 1$) if and only if there exists a strongly connected S-component (S_1, T_1, F_1) with $s \in S_1$ and $M_0(S_1) \leq m$.
(2) There exists a marking $M \in [M_0\rangle$ satisfying $M(s) = m$ ($m \geq 1$) if and only if $M_0(S_1) \geq m$ is true for all strongly connected S-components (S_1, T_1, F_1) with $s \in S_1$. ■

In both (1) and (2), one of the two directions is easy to prove, using the properties of S-components and their derived S-invariants. The nontrivial part of (1) states that the boundedness of s entails the existence of an S-component covering s. The nontrivial part of (2) states that the least bound for the number of tokens on s that can be derived from the S-components can actually be realised by some firing, that is, that there exists a reachable marking placing as many tokens on s as are allowed by the S-components covering s.

As a corollary, it follows that a live FC-system is m-bounded if and only if it is covered by a set of strongly connected S-components with m or less tokens. Consider, as an example, Figure 21. The initial marking shown on the left-hand side is live, and the system is also safe. According to the proposition, it should therefore be covered by strongly connected S-components carrying one token each. There exist two such S-components. One of them is shown in the middle of the figure, the other one is symmetrical.

The strongly connected S-components of a T-system are precisely its simple cycles. Hence a live T-system is m-bounded if and only if there exists a covering by simple cycles which carry m or less tokens. For example, in Figure 22, places s_1 and s_6 are on an S-component with 2 tokens, and they are, moreover, not on

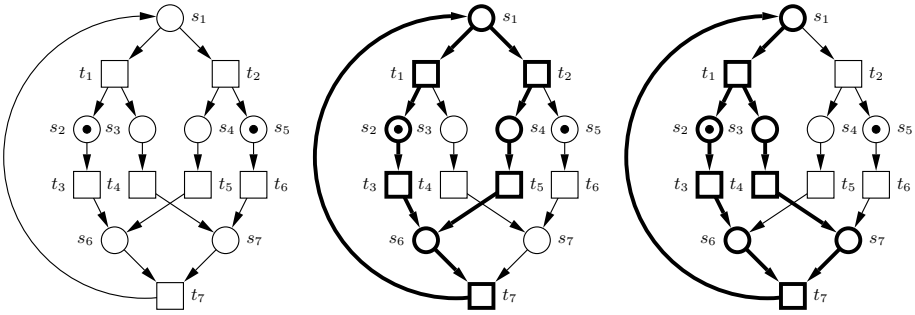


Fig. 21. An initially marked FC-net (l.h.s.); an S-component (middle); a T-component (r.h.s.)

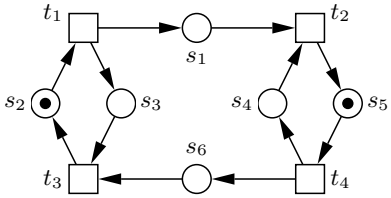


Fig. 22. A 2-bounded T-System

any other (strongly connected) S-component. Hence by Theorem 6 (part (2)⇐), there must be some firing sequence putting two tokens on s_1, and another firing sequence putting two tokens on s_6. Indeed, $t_1t_4t_3t_1$ results in two tokens on s_1, while $t_1t_4t_2t_4$ results in two tokens on s_6.

Well-formed FC-systems also satisfy a T-component covering property, as follows.

Theorem 7. *Covering by T-components*
A live and bounded FC-system N is covered by strongly connected T-components. Moreover, for every strongly connected T-component N_1 in the cover, there exists a reachable marking M such that M, restricted to N_1, is a live and bounded marking of N_1 (as a separate net). ■

As an example, consider Figure 21. The net is covered by two strongly connected T-components, and one of them is shown in bold on the right-hand side of the figure.

There are various ways in which Theorems 6 and 7 can be proved. One can show that in a well-formed FC-net, minimal semipositive S-invariants, minimal nonempty siphons and strongly connected S-components essentially agree with each other and that every place is contained in one of them. One can also make a connection between minimal semipositive T-invariants, minimal reproduction sequences and strongly connected T-components. Alternatively, one can prove one from the other theorem using the duality principle explained in the next section (which would then, in turn, need an independent proof).

3.5 Well-Formedness Criteria, the Duality Theorem, and Net Reductions

The coverability theorems of the previous section and Corollary 4 yield an exact condition for well-formedness, as follows.

Corollary 5. *Well-formedness criterion for FC-nets*
For a plain, weakly connected FC-net N, the following are equivalent:
(i) N is well-formed
(ii) N satisfies (PS) and (PT), and the rank of its incidence matrix is $\leq |CC_N| - 1$. ■

This corollary directly leads to the following duality theorem. Let the *reverse* of a net N be obtained by changing the directions of all arcs, the *dual* by exchanging places and transitions, and the *reverse-dual* by changing directions of all arcs as well as exchanging places and transitions. For example, consider the two nets shown in Figures 13 and 15 which are reproduced in Figure 23. These nets are duals and reverses of each other, and both are self-reverse-dual.

Corollary 6. *Duality theorem for FC-nets*
A plain, weakly connected net is a well-formed FC-net if and only if its reverse-dual is a well-formed FC-net.

Proof: The FC property and conditions (PS),(PT) are invariant with respect to reverse-duality. Moreover, the rank of C (i.e. of N) equals the rank of C^{T} (i.e. of the reverse-dual of N), and the number of clusters is the same in N and in its reverse-dual. The claim then follows with Corollary 5. ■

An almost fully graph-oriented way of characterising well-formed FC-nets can be achieved by *net reductions*. Only three rules are needed. Suppose in the following that (S, T, F) is a plain and weakly connected net.

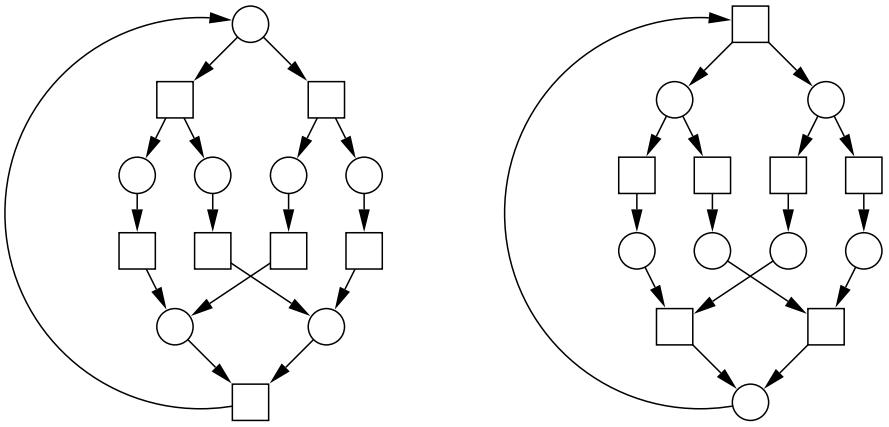


Fig. 23. The nets shown in Figure 15 (l.h.s.) and Figure 13 (r.h.s.)

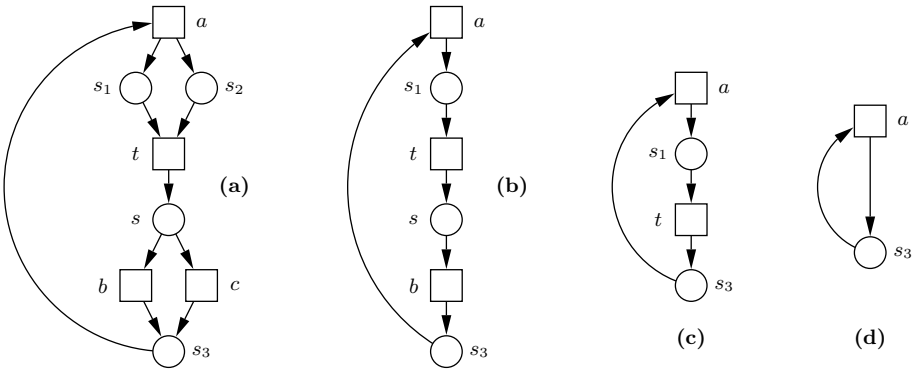


Fig. 24. A sample Petri net reduction

ST-reduction: Suppose $s \in S$ and $t \in T$ such that ${}^\bullet s \neq \emptyset$, $t^\bullet \neq \emptyset$, $s^\bullet = \{t\}$, ${}^\bullet t = \{s\}$, and $({}^\bullet s \times t^\bullet) \cap F = \emptyset$. Then omit s and t and all arrows around s and t, while introducing a new arrow from every $u \in {}^\bullet s$ to every $r \in t^\bullet$.

S-reduction: Suppose a place s is nonnegatively linearly dependent on a set of other places. Then omit s, along with all arrows around it.

T-reduction: Suppose a transition t is nonnegatively linearly dependent on a set of other transitions. Then omit t, along with all arrows around it.

A simple example is shown in Figure 24. Rules are applied in this example as follows:

(a) to (b): S-reduction and T-reduction. The vector for place s_2 is $1\times$ the vector for place s_1, whence s_2 depends linearly and nonnegatively on s_1. Similarly, transition c is $1\times$ transition b. (In this case, the two places and the two transitions actually duplicate each other, which amounts to a special case of the S- (and T-, respectively) reduction rule.)

(b) to (c): ST-reduction with s and b

(c) to (d): ST-reduction with s_1 and t.

Note that at the end (in Figure 24(d)), a loop consisting of a single place and a single transition is obtained. Call this net the *loop net*.

Theorem 8. *Reduction theorem for FC-nets*
A plain, weakly connected FC-net is well-formed if and only if it can be reduced to the loop net by the three reduction rules defined above. ■

3.6 Home States in Free-Choice Nets

The initial marking in Figure 21 is live and safe, but cannot be reproduced by any nonempty firing sequence. As soon as one of the initially activated transitions t_3 or t_6 occur, the initial marking is no longer reachable. The property of being reachable from arbitrary reachable markings is called the *home state property*. In Figure 21, the initial marking is not a home state.

Definition 21. *Home state*
Let $N = (S, T, F, M_0)$ be a marked net. A marking $M \in [M_0\rangle$ is called *home state* or *home marking* if for all $M' \in [M_0\rangle$, $M \in [M'\rangle$ holds true. ■

There are various ways of convincing oneself that M_0, in Figure 21, is not a home state. One possibility is to construct the reachability graph (which is actually depicted on the right-hand side of Figure 7). This graph has a unique 'last' strongly connected component, but M_0 is not contained in it. In fact, *only* M_0 is not contained in it, so that all reachable markings except M_0 are home states.

Another possibility is to use a trap of the net, as follows. Consider in particular the trap $Q = \{s_1, s_3, s_4, s_6, s_7\}$ (cf. Figure 21). Q is token-free initially. However, both t_3 and t_6 put a token on Q, and by the trap property, Lemma 7, Q can never again become empty of tokens. Hence M_0 cannot possibly be a home state. In general:

> If there is a nonempty trap which is token-empty in some marking M and the net is live, then M cannot be a home state.

For live and bounded FC-nets, the converse is also true.

Theorem 9. *Trap theorem for FC-nets*
Let $N = (S, T, F, M_0)$ be a live and bounded FC-system.
M_0 is a home state if and only if all traps $Q \neq \emptyset$ satisfy $M_0(Q) > 0$.

Proof: (*Sketch.*)
Proving ($\Rightarrow$) is easy; the proof was already sketched.
($\Leftarrow$):

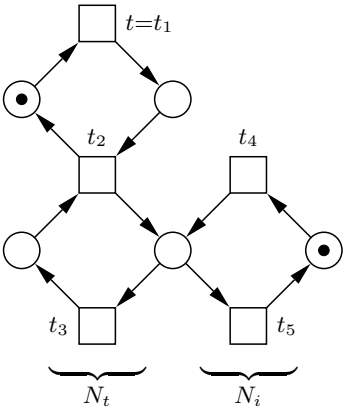
Note first that a marking may be live and safe even if every strongly connected T-component contains a token-free cycle, that is, even if *no* T-component is live when seen as a separate T-system. This is indeed the case in Figure 21. The T-component shown there has the token-free cycle $\{s_1, t_1, s_3, t_4, s_7, t_7\}$. The other strongly connected T-component also has a token-free cycle.

If a strongly connected T-component has no token-free cycle, we call it *activated*. Taken in isolation, an activated T-component is a live T-system, to which Theorem 5 applies. Inside an FC-system, transitions of an activated T-component can always be chosen by the free choice property in favour of others that would effect token loss on it. It is known that every marking of a strongly connected, live T-system is a home state. Therefore, if t lies inside an activated T-component in a live FC-net, and if $M\,[t\rangle M'$, then $M \in [M'\rangle$; that is, the firing of t can be reversed. This argument extends inductively to firing sequences. If, in a firing sequence $M_0\,[t_1\rangle M_1\,[t_2\rangle M_2 \ldots M_{n-1}\,[t_n\rangle M_n$, every transition t_i is inside some activated T-component, then $M_0 \in [M_n\rangle$.

Now suppose that $M_0\,[t\rangle M$. We want to prove that $M_0 \in [M\rangle$.

If t is inside some strongly connected T-component which is activated at M_0, then by the argument just given, $M_0 \in [M\rangle$, and we are done.

However, there might not be *any* activated T-components containing t. We show that nevertheless, M_0 can be reached from M as follows:



$M_0[t_1\rangle M$

$M[t_4t_3t_2t_3t_1t_2t_5\rangle M_0$

Fig. 25. An FC-system whose initial state is a home state; transition t is not inside an activated T-component

- Start with *some* initially activated strongly connected T-component N_i. From the fact that all nonempty traps are marked initially, it can be proved that an initially activated, strongly connected T-component exists.
- Using the covering theorem, pick any T-component N_t containing t.
- Now execute activated T-component(s) as much as possible, but without firing either t or any other transition in its conflict cluster. Do this in such a way that tokens are 'moved towards' N_t. This can be achieved by a suitable allocation, as in the liveness theorem. Say, $M_0\,[\tau\rangle\widetilde{M_0}$ with a maximal sequence τ satisfying this property. It can be shown that this can be done in such a way that at $\widetilde{M_0}$, N_t is activated and t is still enabled. Thus $M_0\,[\tau t\rangle$ and also $M_0\,[t\tau\rangle$.
- Let $M_0\,[\tau\rangle\widetilde{M}\,[t\rangle\widetilde{M}$ and also $M_0\,[t\rangle M\,[\tau\rangle\widetilde{M}$. Then $\widetilde{M}\,[\tau'\rangle M_0$ with some sequence τ', because both τ and the subsequent firing of t can be reversed (for they all take place within activated T-components). But then also $M_0\,[t\rangle M\,[\tau\rangle\widetilde{M}\,[\tau'\rangle M_0$, showing that $M_0 \in [M\rangle$. ■

An example explaining this proof is shown in Figure 25. Suppose $M_0[t_1\rangle M$. We want to show that M_0 can be reached from M. Transition $t = t_1$ is inside the T-component N_t shown in the figure, but N_t is not activated. However, there is another, initially activated, T-component N_i whose transitions can be executed in a reversible way. A maximal sequence activating N_t and not containing t itself is $\tau = t_4t_3$ which can be fired from M_0 and also from M (note that t_3 was chosen rather than t_5). The sequence τ' constructed in the last step of the proof is $\tau' = t_2t_3t_1t_2t_5$. Hence M_0 can be reached from M by $t_4t_3t_2t_3t_1t_2t_5$. Note how τ' 'undoes' first t, by t_2t_3; then t_3, by t_1t_2; and then t_4, by t_5.

Corollary 7. *Confluence*

Let $N = (S, T, F, M_0)$ be a live and bounded FC-system and let $M_1, M_2 \in [M_0\rangle$.

Then $[M_1\rangle \cap [M_2\rangle \neq \emptyset$. ■

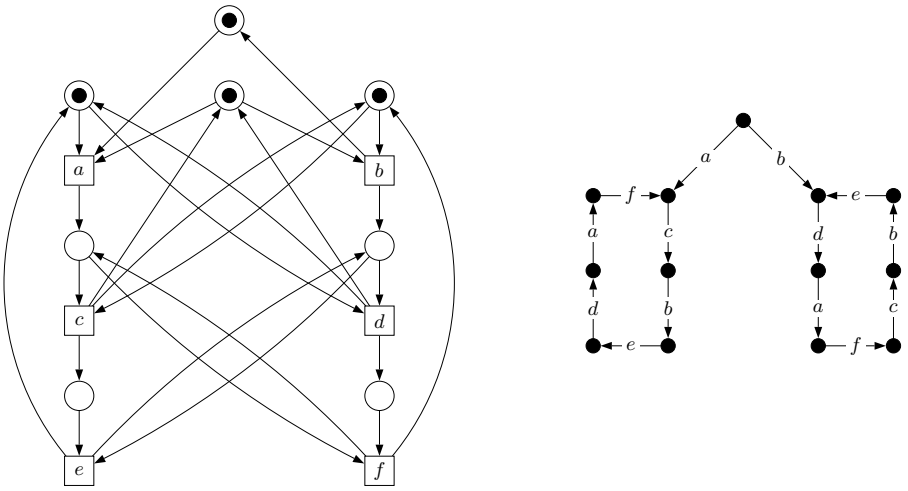


Fig. 26. A live and 2-bounded net without home states (l.h.s.); its reachability graph (r.h.s.)

Corollary 8. *Existence of home states*
Let $N = (S, T, F, M_0)$ be a live and bounded FC-system. There exists a home state $M \in [M_0\rangle$. ■

A firing sequence containing every transition at least once necessarily generates a home state, since every trap $Q \neq \emptyset$ has at least one incoming transition. Such a firing sequence exists by liveness. The free-choice property is important. If it is omitted, we can find counterexamples such as the one shown in Figure 26.

The next result about blocking markings shows that it is in general possible to find home states, even without necessarily firing *all* transitions.

Definition 22. *Blocking marking*
Let N be a plain net with initial marking M_0 and let $K \in CC_N$ be a conflict cluster. A *blocking marking for* K is a marking $M_K \in [M_0\rangle$ such that every transition in K is enabled by M_K and no other transitions are enabled by M_K. ■

Theorem 10. *Existence and uniqueness of blocking markings*
Let $N = (S, T, F, M_0)$ be a plain, live and bounded FC-net and let $K \in CC_N$. There exists a unique blocking marking $M_K \in \mathcal{E}(M_0)$ associated with K.

Proof: (*Sketch.*)
Consider the net which remains if the T-component shown in bold on the right-hand side of Figure 21 is erased. Note that it is an acyclic T-system with a unique *starting transition*, t_2 ('starting' is meant in the sense of the flow relation). It so happens that one can always find a minimal cover of N with some strongly

connected T-components in such a way that taking away a carefully chosen one of them makes this is true in general.

Let t_{in} be the starting transition of such a subnet. It can be shown that in any blocking marking for the cluster of t_{in}, there exists a token-free path from t_{in} to any other transition in the subnet. If such a situation is given in some T-system, then it can be shown that the marking is unique (on the subnet). This yields a lever in order to prove the theorem (in particular, the uniqueness of the blocking marking) by induction on the number of T-components in a minimal strongly connected T-component covering of N. ■

Existence and uniqueness of a blocking marking implies that any such marking is a home state. Thus, in a weakly connected FC-system, a home state can be reached by the following procedure: (a) fix some enabled transition t (then by the FC property, all transitions of the cluster $({}^\bullet t)^\bullet$ of t are enabled); (b) fire transitions in $T \setminus ({}^\bullet t)^\bullet$ until it is no longer possible.

3.7 Realisability and Reachability Analysis

In section 2.1, it was emphasised that the marking equation or the marking inequality are necessary, but not sufficient for realisability or reachability. In the present section, this predicament will be discussed in more detail. It will be seen that nevertheless, under certain conditions, one may get some sort of converse of the firing lemma (Lemma 2).

Before starting the discussion, let us reconsider the non-structurally-live and non-structurally-bounded nets from section 1.7, reproduced here in Figure 27(a) and (b). Consider some elementary directed cycle in a Petri net. Any elementary directed path which starts at some place of the cycle and ends at some transition of the cycle but does not touch the cycle at any point in between is called a PT-*handle* (for this cycle). The notion of TP-*handle* is defined symmetrically. Note that there exist PT-handles in Figure 27(a) and TP-handles in 27(b).

Intuitively, a PT-handle is detrimental for liveness, because some tokens needed for firing the cycle's transitions could 'get lost' on it, such as in Figure 27(a). TP-handles, on the other hand, seem to be detrimental for boundedness, because

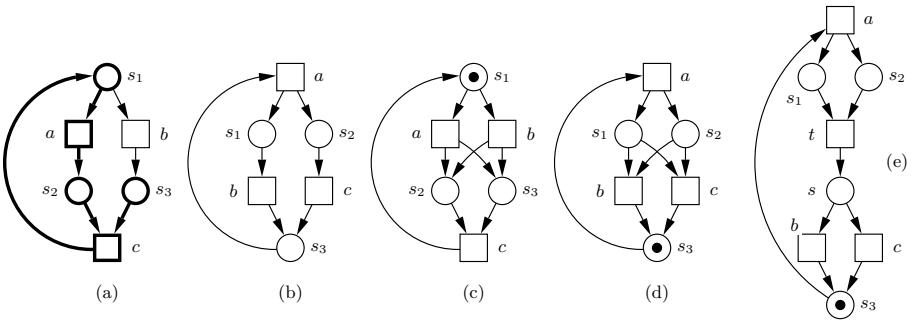


Fig. 27. Handle examples

tokens not needed for a cycle's liveness may be produced indefinitely, such as in Figure 27(b). However, one should be careful because not every PT-handle or TP-handle leads to non-liveness or to non-boundedness. The systems shown in Figure 27(c) and (d) contain handles, but they are perfectly live and bounded (even safe) FC-systems. Note that (c) is also an fc-system while (d) is not.

In order to state a realisability theorem, we need to define subnets generated by T-vectors. Let $y \in \mathbb{N}^T$ be any semipositive T-vector and consider the *support* of y, which is defined as $supp(y) = \{t \in T \mid y(t) > 0\}$. The *subnet generated by* y, N_y, is defined as the subnet $N(T_y, S_y)$ where $T_y = supp(y)$ and S_y is the set of all places which are either input or output places of $supp(y)$. For example, the subnet generated by the T-vector $(2, 0, 1)$ is shown in bold in Figure 27(a).

Theorem 11. *First realisability criterion*

Let $N = (S, T, F)$ be a plain, pure net without PT-handles and let $y \in \mathbb{N}^T$ be a semipositive T-vector.

Then y is realisable from a marking M if and only if $M + C{\cdot}y \geq 0$ and N_y has no token-free (under M, restricted to N_y) nonempty siphons.

Proof: (*Sketch.*)
The problematic direction is ($\Leftarrow$). This can be proved (a) for FC-nets where every place has at most two output transitions, then (b) for arbitrary FC-nets by reducing them to (a), and finally (c) for arbitrary nets by reducing them to FC-nets. ■

Part (c) of the proof relies on a construction replacing every arc from a place to a transition by a sequence arc-transition-arc-place-arc. Such a construction transforms every net into an FC-net; it works for the present purpose, but not for all purposes, as it may, e.g., introduce new deadlocks.

In Figure 27(a) with an initial token on s_1, the vector (1 1 1) is not realisable even though it satisfies the marking inequality $M_0 + C{\cdot}y \geq 0$ and there are no (nonempty) token-free siphons. This shows that the premise of there not being any PT-handles is necessary for the theorem to hold. By duality (more precisely: considering the reverse net), the following theorem is a corollary:

Theorem 12. *Second realisability criterion*

Let $N = (S, T, F)$ be a plain, pure net without TP-handles and let $y \in \mathbb{N}^T$ be a semipositive T-vector.

Then y is realisable from a marking M if and only if $M + C{\cdot}y \geq 0$ and N_y has no token-free (under $M + C{\cdot}y$, restricted to N_y) nonempty traps. ■

Both theorems can be turned into exact reachability criteria for nets without PT-handles or nets without TP-handles. Such a class is given by live and bounded fc-systems, since it is known that such nets do not have TP-handles. Consider Figure 27(d). It shows a live and bounded FC-system with TP-handles. A systematic transformation of it into an fc-system would just insert between a conflict cluster U and its preset ${}^\bullet U$ a single transition followed by a single place, provided that $|{}^\bullet U| \geq 2$. The result for Figure 27(d) is shown in part (e) of the figure. The TP-handles have disappeared.

The reachability criterion for live and bounded fc-systems can be formulated as follows:

- Given: a live and bounded fc-system (S, T, F, M_0) and a marking $M \in \mathbb{N}^S$.
- Solve, if possible, the following system of linear (in)equations for the unknown vector y:

$$\begin{aligned} & y \in \mathbb{N}^T \\ & M_0 + C \cdot y \;=\; M \end{aligned}$$

  under the further constraint that N_y has no token-free trap under M.
- If this is possible, $M \in [M_0\rangle$, otherwise $M \notin [M_0\rangle$.

The correctness of this procedure follows from Theorem 12. The theorems are applicable to other classes of systems as well.

3.8 Bibliographical Remarks and Further Reading

T-systems have traditionally been called *marked graphs* [CHEP71] or *synchronisation graphs* [GL73]. The definitive book on the structure theory of free-choice systems is [DE95], by Jörg Desel and Javier Esparza, which contains several of the results and arguments described in this section, such as the Commoner/Hack liveness theorem [Hac72], the home state theorems [BV84, BDE92], the coverability and duality theorems, and the reduction theorem. Since the publication of this meritorious piece of work, further structural results in a similar vein have been discovered, e.g.: [Esp98] (NP-completeness of reachability in live and safe FC-systems); the blocking theorem described in section 3.6 [GHM03, Weh10]; general reachability criteria as described in section 3.7, which are due to Hideki Yamasaki, Jeng S. Huang and Tadao Murata [YHM01], with related results in [LR94, MM98, YY03]; and the proof, by Joachim Wehler [Weh09], of an old conjecture on a subclass of free-choice systems by Hartmann Genrich and P.S. Thiagarajan [GT84], making a connection to the almost equally old notion of frozen tokens [BM85]; not to mention many generalisations and extensions of these results, e.g. by the active research group around Manuel Silva [RTS98]. The literature also offers generalisations of definitions and results for arc-weighted T-systems [TCCS92] and for arc-weighted FC-systems [TS96].

4 Conflict Structure of Petri Nets

In a T-system, tokens cannot be removed from a place but by the – unique, if any – output transition of such a place. This implies what at the end of section 1.2 has somewhat loosely been called the absence of conflicts, or persistency. Symmetrically, in S-systems, transitions cannot be hindered from firing except by the – unique, if existing – input place of such a transition. This can loosely be called the absence of synchronisation, or communication-freeness.

Both notions, the absence of conflicts and the absence of synchronisation, give rise to a variety of structural constraints that partially overlap with those

considered in the previous sections. For example, we might relax the notion of a T-system by requiring only $|s^\bullet| \leq 1$ or $|s^\bullet| = 1$, but not necessarily also $|{}^\bullet s| \leq 1$, for all $s \in S$. Such a class of nets might be called *place-output-nonbranching*. Symmetrically, we might relax the notion of an S-system by requiring only $|{}^\bullet t| \leq 1$ or $|{}^\bullet t| = 1$, not necessarily also $|t^\bullet| \leq 1$, for all $t \in T$. Such a class of nets might be called *transition-input-nonbranching*.

In the present section, we shall concentrate on restrictions related to persistency and to the absence of conflicts. Amongst others, we examine the class of place-output-nonbranching Petri nets, just called *output-nonbranching nets*, for short. Symmetrical restrictions related to the absence of synchronisation will not be examined in detail.

In the last part of the paper, a property known as *separability* is studied. This property indicates that a system can be viewed as a superimposition of independent subsystems, and it is a desirable feature in some applications. It turns out that in persistent systems, some (partly structural) conditions guaranteeing separability can be given.

We continue to assume that every net is plain.

4.1 A Hierarchy of Petri Nets without Conflicts

There is a surprising variety of classes of nets, all of which could (more or less) be called 'conflict-free'. By historical developments, the privilege of bearing the actual name, '*conflict-free nets*', has been bestowed onto one of these classes. The next definition introduces this class, as well as several related ones.

Definition 23. *Output-nonbranching, conflict-free, and persistent nets*
Let $N = (S, T, F, M_0)$ be a net with an initial marking.

- N is called *output-nonbranching* (ON) if all places s satisfy $|s^\bullet| \leq 1$.
- N is called *conflict-free* (CF) if all places s satisfy $|s^\bullet| > 1 \Rightarrow s^\bullet \subseteq {}^\bullet s$.
- N is called *behaviourally conflict-free* (BCF) if for any two transitions $t, t' \in T$ with $t \neq t'$ and for every $M \in \mathcal{E}(M_0)$, if $M\,[t\rangle$ and $M\,[t'\rangle$ then ${}^\bullet t \cap {}^\bullet t' = \emptyset$.
- N is called *binary-conflict-free* (BiCF) if for any two transitions $t, t' \in T$ with $t \neq t'$ and for every $M \in \mathcal{E}(M_0)$, if $M\,[t\rangle$ and $M\,[t'\rangle$ then $\forall s \in S\colon M(s) \geq F(s,t)+F(s,t')$.
- A transition $t \in T$ is called *persistent*, if for every reachable marking $M \in \mathcal{E}(M_0)$, and for every transition $t' \in T$ with $t \neq t'$, if $M\,[t\rangle$ and $M\,[t'\rangle$ then $M\,[tt'\rangle$ and $M\,[t't\rangle$. N is called *persistent* if every transition is persistent.
- A transition $t \in T$ is called *weakly persistent*, if for every reachable marking $M \in \mathcal{E}(M_0)$ and for every sequence $\sigma \in T^*$, if $M\,[t\rangle$ and $M\,[\sigma t\rangle$ then $M\,[t\sigma'\rangle$ for some permutation σ' of σ. N is called *weakly persistent* if every transition is weakly persistent. ■

Whether a net is output-nonbranching or conflict-free depends only on its structure. These properties can be checked without necessarily constructing the reachability graph. The other properties can be detected on the reachability graph.

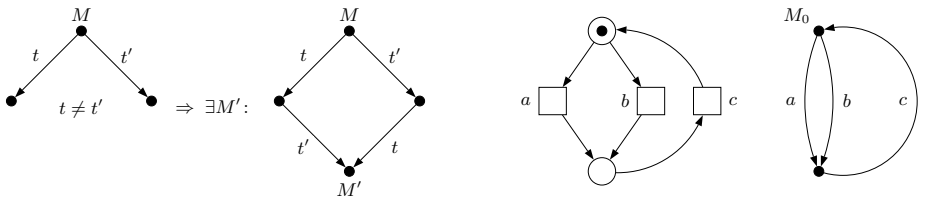


Fig. 28. Illustration of persistency (l.h.s.); a non-persistent net and its reachability graph (r.h.s.)

For instance, behavioural conflict-freeness can be checked as follows. Whenever a vertex is encountered from which two or more arcs labelled t and t', respectively, emanate, we check any pair of such arcs for the property ${}^\bullet t \cap {}^\bullet t' = \emptyset$, which can be read off the net. In order to check the persistency of a transition t, it is sufficient to check the property indicated in Figure 28, for every transition $t' \neq t$ and for every vertex M in the reachability graph.

The BiCF condition, $\forall s \in S\colon M(s) \geq F(s,t)+F(s,t')$, indicates that t and t' are *concurrently enabled*. We shall therefore use the shorthand $M[\{t,t'\}\rangle$ in order to denote $\forall s \in S\colon M(s) \geq F(s,t)+F(s,t')$. The difference between BCF and BiCF (and persistency) can be seen on Figure 29.

From Definition 23, one gets the hierarchy shown in Figure 30. Thus, the class of T-systems is the smallest class under consideration while the class of weakly persistent systems is the largest class (i.e., all others lie inside). Actually, it is hard to call the class of weakly persistent systems 'conflict-free' since it contains systems that clearly exhibit conflicts in the intuitive sense, such as the one on the right-hand side of Figure 28. Nevertheless, as we will show in the next section, weakly persistent systems do enjoy some properties normally associated with persistent and conflict-free systems.

Proof sketch of the implications shown in Figure 30:
Every T-system is also ON: this follows directly from the definitions.

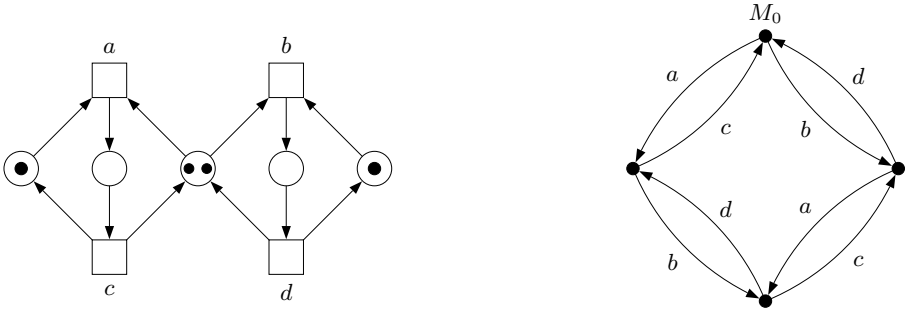


Fig. 29. A net which is persistent and BiCF but not BCF (l.h.s.), and its reachability graph (r.h.s.)

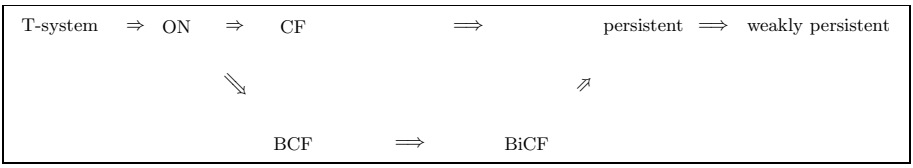


Fig. 30. A hierarchy of conflict-free and persistent Petri nets

Every ON system is CF: the CF condition is trivially true for ON systems.
Every CF system is persistent: if ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and $t \neq t'$, then for any $s \in {}^\bullet t \cap {}^\bullet t'$, $|s^\bullet| > 1$. Then by the CF condition, both $t \in {}^\bullet s$ and $t' \in {}^\bullet s$, so that the occurrence of either of them cannot disable the other one. (Note that this argument is invalid if arc weights can be greater than 1.)
Every ON system is BCF: for ON systems, $t \neq t'$ already suffices to imply ${}^\bullet t \cap {}^\bullet t' = \emptyset$.
Every BCF system is BiCF: suppose that M enables both t and t' with $t \neq t'$; by the BCF property, ${}^\bullet t \cap {}^\bullet t' = \emptyset$, and then, $M[\{t, t'\}\rangle$ is necessarily true because of $M[t\rangle$ and $M[t'\rangle$.
Every BiCF system is persistent: $M[\{t, t'\}\rangle$ implies that both $M[tt'\rangle$ and $M[t't\rangle$.
Every persistent system is weakly persistent: this follows from Keller's theorem, to be stated below. ■

Some of the implications of Figure 30 can be reversed under (relatively) weak conditions. They were described in the figure by short arrows. Implications indicated by long arrows cannot be reversed so nicely. To see this, we examine more closely the case that $M[tt'\rangle$ and $M[t't\rangle$ for some marking M and two transitions $t \neq t'$. Because of its shape in the reachability graph, such a situation is called a *diamond*. A diamond comes in two varieties. If $t \neq t'$ and $M[\{t, t'\}\rangle$, then it is called a *concurrent diamond*. (Note that both $M[tt'\rangle$ and $M[t't\rangle$ are implied by $M[\{t, t'\}\rangle$). If $t \neq t'$ and $M[tt'\rangle$ and $M[t't\rangle$ but $\neg M[\{t, t'\}\rangle$, then the diamond is called a *conflicting diamond*. Figure 31 shows the difference. Self-loops are necessary for the existence of conflicting diamonds. If a Petri net is free of self-loops, all diamonds are concurrent.

The next proposition shows that properties ON and CF are essentially equivalent to each other, and both properties are also close to T-systems. Moreover, property BiCF is almost equivalent to persistency, except for the difference

concurrent diamond conflicting diamond

Fig. 31. A concurrent diamond (l.h.s.) and a conflicting diamond (r.h.s.)

between the two types of diamonds just explained. In particular, in self-loop-free Petri nets, BiCF is the same as persistency.

Proposition 6. *Some relationships between classes of systems without conflicts*

1. For every CF net N with initial marking M_0, an ON net N' with initial marking M_0' can be constructed such that the two reachability graphs are isomorphic.
2. Every live and bounded ON system is a T-System.
3. A net is BiCF if and only if it is persistent and there is no conflicting diamond.

Proof: To prove 1., consider an arbitrary place s with $|s^\bullet| > 1$. By the CF property, $s^\bullet \subseteq {}^\bullet s$, that is, s is a side condition for every output transition in $s^\bullet$. Replace s by $|s^\bullet|$ new places which are marked and connected as s, except that a side condition connects it to only one (not all) of the $|s^\bullet|$ output transitions. The reachability graph of this new net is isomorphic to the old one. The construction can be repeated until there are no more places s with $|s^\bullet| > 1$.

To prove 2., let $N = (S, T, F, M_0)$ be a live and bounded ON system. By Proposition 5, every weakly connected component of N is strongly connected. Since N is an ON net, it is also an FC-net. Thus it is covered by S-components by Theorem 6, and hence, also structurally bounded because it is covered by a positive S-invariant. By Proposition 1 and Farkas' lemma (Lemma 3), there is, therefore, no vector $y \in \mathbb{N}^{|T|}$ with $C \cdot y > 0$. Because N is an ON net and is covered by cycles, $C \cdot y \geq 0$ where y is the all-ones T-vector 1. Suppose, for a contradiction, that N is not a T-net. Then $C \cdot y \neq 0$, because there is at least one place with more than one input transition, contradicting the fact, just proved, that no such y exists.

To prove 3.($\Rightarrow$), we have already seen that BiCF implies persistency. BiCF also implies the absence of conflicting diamonds: if there is such a diamond with M, t and t', then the BiCF property is violated with the same transitions at the same marking. To prove 3.($\Leftarrow$), assume $M[t\rangle$ and $M[t'\rangle$ with $t \neq t'$. By persistency, we get the diamond $M[tt'\rangle$ and $M[t't\rangle$. By the absence of conflicting diamonds, this is a concurrent diamond; thus $M[\{t, t'\}\rangle$. ■

It is perhaps illuminating to compare this Petri net hierarchy with the one defined in the previous section. T-systems (marked graphs) and ON-nets are both free-choice and persistent. But Proposition 6 notwithstanding, there exist CF nets in the sense of Definition 23 which are not FC in the sense of Definition 18, and conversely. The same is true for BCF nets. In fact, S-systems are free-choice but, in general, not even persistent. An example can be found on the right-hand side of Figure 28. Persistent nets are considerably less well-behaved than T-systems. For instance, even if they are strongly connected, there may be reproducing T-vectors which are different from multiples of 1 (as an example, see Figure 33 below). While in a T-system, every live marking is a home state, there exist live and bounded persistent systems whose initial marking is not a home state. The

net shown in Figure 16 can be turned into an example by putting tokens on s_2, s_5 and r_1.

4.2 Weak Persistency and Semilinearity

Some times, the reachability set of a Petri net has a *semilinear* representation.

Definition 24. *Semilinear sets*
A set $W \subseteq \mathbb{N}^n$ is called *linear* if there are vectors $b, p_1, \ldots, p_\ell \in \mathbb{N}^n$ (with $\ell \in \mathbb{N}$) such that

$$W = \{b + \sum_{i=1}^{\ell} n_i \cdot p_i \mid n_i \in \mathbb{N}\}.$$

A set $W \subseteq \mathbb{N}^n$ is called *semilinear* if it is the finite union of linear sets. ■

The vectors named b are the *bases*, and the vectors named p are the *periods*.

For instance, the reachability set in Equation (1) of section 1.2 has a semilinear representation as follows:

$$\begin{array}{l}
\{\ (0\ \ 1\ \ 2)^{\mathsf{T}}\ \} \cup \{\ (0\ \ 0\ \ 4)^{\mathsf{T}}\ \} \\
\cup\ \{\ (1\ \ 0\ \ 3)^{\mathsf{T}} + n_1 \cdot (0\ \ 0\ \ 1)^{\mathsf{T}} \mid n_1 \in \mathbb{N}\ \} \cup \{\ (2\ \ 0\ \ 2)^{\mathsf{T}} + n_1 \cdot (0\ \ 0\ \ 1)^{\mathsf{T}} \mid n_1 \in \mathbb{N}\ \} \\
\cup\ \{\ (3\ \ 0\ \ 1)^{\mathsf{T}} + n_1 \cdot (0\ \ 0\ \ 1)^{\mathsf{T}} \mid n_1 \in \mathbb{N}\ \} \cup \{\ (4\ \ 0\ \ 0)^{\mathsf{T}} + n_1 \cdot (1\ \ 0\ \ 0)^{\mathsf{T}} + n_2 \cdot (0\ \ 0\ \ 1)^{\mathsf{T}} \mid n_1, n_2 \in \mathbb{N}\ \} \\
\cup\ \{\ (1\ \ 1\ \ 1)^{\mathsf{T}} + n_1 \cdot (0\ \ 0\ \ 1)^{\mathsf{T}} \mid n_1 \in \mathbb{N}\ \} \cup \{\ (2\ \ 1\ \ 0)^{\mathsf{T}} + n_1 \cdot (1\ \ 0\ \ 0)^{\mathsf{T}} + n_2 \cdot (0\ \ 0\ \ 1)^{\mathsf{T}} \mid n_1, n_2 \in \mathbb{N}\ \}
\end{array}$$

with eight bases and two periods combined in appropriate ways. In general, however, there is no such easy representation of the reachability set, since it is known that there exist nets whose reachability sets are not semilinear.

One of the seminal results about persistent nets (and later also weakly persistent nets) is that they always have semilinear reachability sets. Even more, weak persistency is decidable, and if the decision is positive, the semilinear reachability set can be constructed.

For the proof a simple conclusion needs to be drawn from the definition of weak persistency. If for some weakly persistent net $N = (S, T, F, M_0)$ there are $\sigma, \sigma' \in T^*$ with $M\,[\sigma\rangle$, $M\,[\sigma'\rangle$ and $\mathcal{P}(\sigma) \geq \mathcal{P}(\sigma')$ there is also some σ'' with $M\,[\sigma'\sigma''\rangle$ and $\mathcal{P}(\sigma) = \mathcal{P}(\sigma'\sigma'')$. This follows directly by induction over the length of σ'. For persistent nets this conclusion can be strengthened; this will be discussed in the next section.

Theorem 13. *Weak persistency is decidable*
Let $N = (S, T, F, M_0)$ be a Petri net. It is decidable if N is weakly persistent. Furthermore, weakly persistent nets have semilinear reachability sets.

Proof: (*Sketch.*) Construct a set EM of extended markings $(x, M) \in \mathbb{N}^T \times \mathbb{N}^S$ where $M_0\,[\sigma\rangle M$ with $\mathcal{P}(\sigma) = x$ holds. The construction of EM starts with $EM = \{(0, M_0)\}$, then consecutively some $(x + 1 \cdot t, M')$ is added to EM if $(x, M) \in EM$

and $M\,[t\rangle M'$. Take some stage k where $EM_k = \{(x_1, M_1), \ldots, (x_k, M_k)\} \subseteq EM$ has been computed so far. Define nonnegative difference sets D_i for $1 \leq i \leq k$ by

$$D_i = \{(x_j - x_i, M_j - M_i) \,|\, 1 \leq j \leq k,\ M_j \geq M_i\}.$$

Then,

$$S_k = \bigcup_{1 \leq i \leq k} \left\{ (x_i, M_i) + \sum_{d \in D_i} n_d d \;\middle|\; n_d \in \mathbb{N} \right\}$$

is a semilinear set whose projection on the second component approaches $\mathcal{E}(M_0)$ from below with increasing k. Two decidable formulae can be built using S_k, formula A checking if for all $(x, M) \in S_k$ with $M\,[t\rangle M'$ also $(x + 1{\cdot}t, M') \in S_k$. If so, S_k is complete ($S_k = EM$) and its projection to the markings is the set $\mathcal{E}(M_0)$ which is also semilinear. Formula B checks if for any pair $(x, M), (x', M') \in S_k$ with $x \leq x'$ no transition $t \in T$ with $x(t) < x'(t)$ is enabled. This would contradict the above conclusion from weak persistency. If N is weakly persistent it can be shown that the number of different sets D_i is finite even for $k \to \infty$, i.e. for the complete set $EM = \lim_{k\to\infty} S_k$. The set of all (x_i, M_i) with the same D_i may be infinite, but the minimal elements of this set suffice when building $\lim_{k\to\infty} S_k$ and by Dickson's Lemma there are only finitely many of those. We can conclude EM is semilinear then, so at some finite stage k either formula A or B must hold, deciding if N is weakly persistent or not. If N is weakly persistent, its reachability set is the projection of the final S_k to its second component. ■

Weakly persistent nets share with persistent nets the property of having semilinear reachability sets. However, they do not share the property of, intuitively, being 'nets without conflict'. Consider the net shown on the right-hand side of Figure 28. This net is weakly persistent, but it exhibits a conflict between a and b in its initial state.

4.3 Keller's Theorem

A seminal result about persistent Petri nets (which does not hold for weakly persistent nets in general) is based on the notion of the *residue* of a sequence τ of transitions with respect to another sequence σ, denoted by $\tau \overset{\bullet}{-} \sigma$. By definition, $\tau \overset{\bullet}{-} \sigma$ is what is left of τ after cancelling successively all symbols from σ (if possible), read from left to right. Formally, $\tau \overset{\bullet}{-} \sigma$ can be defined by induction on the length of σ:

$$\begin{aligned}
\tau \overset{\bullet}{-} \varepsilon &= \tau \\
\tau \overset{\bullet}{-} t &= \begin{cases} \tau, \text{ if there is no transition } t \text{ in } \tau \\ \text{the sequence obtained by erasing the leftmost } t \text{ in } \tau, \text{ otherwise} \end{cases} \\
\tau \overset{\bullet}{-} (t\sigma) &= (\tau \overset{\bullet}{-} t) \overset{\bullet}{-} \sigma.
\end{aligned}$$

We now formalise that two firing sequences are permutations of each other from a marking. Two sequences $\sigma \in T^*$ and $\sigma' \in T^*$ are said to arise from each other

by a *transposition from* M if both are activated at M and if they are the same, except for the order of an adjacent pair of transitions, thus:

$$M[\sigma\rangle \quad \text{and} \quad M[\sigma'\rangle \quad \text{and} \quad \sigma = t_1 \dots t_k t t' \dots t_n \quad \text{and} \quad \sigma' = t_1 \dots t_k t' t \dots t_n.$$

Essentially, this means that σ and σ' are the same except for some (not necessarily concurrent) diamond reached after $t_1 \dots t_k$. Two sequences σ and σ' are said to be *permutations of each other from* M (written $\sigma \equiv_M \sigma'$) if they are both activated at M and if they arise out of each other through a sequence of transpositions from M.

Theorem 14. *Keller's theorem*

Let (S, T, F, M_0) be a persistent Petri net. Let τ and σ be two firing sequences activated at some reachable state $M \in \mathcal{E}(M_0)$. Then $\tau(\sigma \stackrel{\bullet}{-} \tau)$ and $\sigma(\tau \stackrel{\bullet}{-} \sigma)$ are also activated from M, and $\tau(\sigma \stackrel{\bullet}{-} \tau) \equiv_M \sigma(\tau \stackrel{\bullet}{-} \sigma)$. Furthermore, the marking reached after $\tau(\sigma \stackrel{\bullet}{-} \tau)$ equals the marking reached after $\sigma(\tau \stackrel{\bullet}{-} \sigma)$.

Proof: (*Sketch.*) By induction on the length of τ. If $\tau = \varepsilon$, both $\tau(\sigma \stackrel{\bullet}{-} \tau)$ and $\sigma(\tau \stackrel{\bullet}{-} \sigma)$ are equal to σ, and the result follows directly from the premise that σ is activated at M, the definition of $\equiv_M$, and persistency. If $\tau = t\tau'$, two cases can be distinguished: σ does not contain t or σ contains t, i.e., $\mathcal{P}(\sigma)(t) = 0$ or $\mathcal{P}(\sigma)(t) > 0$, respectively. In either case, after some manipulations involving permutations of sequences in persistent nets, the induction hypothesis yields the desired result. ■

Note that part of this theorem is a confluence statement. That is, if $M_0 [\sigma\rangle M$ and $M_0 [\tau\rangle M'$, then $\mathcal{E}(M) \cap \mathcal{E}(M') \neq \emptyset$. The other part of the theorem asserts that Parikh vectors of sequences leading to a common successor marking of two reachable markings can actually be computed explicitly, using residues.

Using Keller's theorem, we can easily prove that persistency implies weak persistency. Suppose that N is persistent and that some reachable marking M enables both t and σt. By Keller's theorem, M also enables $t((\sigma t) \stackrel{\bullet}{-} t)$. But $\sigma' = (\sigma t) \stackrel{\bullet}{-} t$ has the same Parikh vector as σ by the definition of $\stackrel{\bullet}{-}$. Hence both $M[\sigma t\rangle$ and $M[t\sigma'\rangle$, with $\mathcal{P}(\sigma t) = \mathcal{P}(t\sigma')$. Again by Keller's theorem, $\sigma t \equiv_M t\sigma'$. Thus N is also weakly persistent.

4.4 Cycle Decompositions, *k*-Nets, and Separability

In this section, several recent results about persistent Petri nets will be described. Theorem 14, in combination with other structural Petri net techniques, is used all over their proofs. We will not sketch these proofs, but illustrate the properties of the definitions and the statements of the results by means of examples.

For the next result, we introduce the concept of a *smallest cycle*. Consider Figure 29. The reachability graph shown on the right-hand side has a cycle $M_0[abcd\rangle M_0$ which is elementary in the sense of section 1.3. However, there is also a non-empty cycle $M_0[ac\rangle M_0$ which has a smaller Parikh vector. Note that both $\mathcal{P}(abcd)$ and $\mathcal{P}(ac)$ are T-invariants, by Lemma 5, showing that the former

is not a minimal one. Inspired by this example, call a cycle in the reachability graph, starting at some marking M, *smallest* (around M) if there is no non-empty cycle around M which has a smaller Parikh vector. Thus, every smallest cycle is elementary, but the converse need not be true. Smallest cycles correspond to minimal T-invariants.

Theorem 15. *Decomposing cycles of bounded, reversible, and persistent nets*
Let $N = (S, T, F, M_0)$ be a bounded, reversible, and persistent Petri net. There exists a finite set $\{X_1, \ldots, X_n\}$ of semipositive T-invariants such that they are transition-disjoint and every cycle $M[\rho\rangle M$ in the reachability graph of N can be decomposed, up to permutations, to some sequence

$$M[\rho_1\rangle M[\rho_2\rangle M \ldots [\rho_n\rangle M$$

of cycles with all Parikh vectors $\mathcal{P}(\rho_i)$ in $\{X_1, \ldots, X_n\}$. Moreover, $\{X_1, \ldots, X_n\}$ can be chosen as the set of Parikh vectors of smallest cycles through any fixed reachable marking of N. ■

To appreciate the relevance of transition-disjointness, reconsider Figure 29. The reachability graph shown there has two transition-disjoint cycles, $(ac)^*$ and $(bd)^*$, which are executable from M_0. They correspond to two realisable, minimal, transition-disjoint T-invariants, $(1\,0\,1\,0)^\mathsf{T}$ and $(0\,1\,0\,1)^\mathsf{T}$. From every state and for each one of these T-invariants, a cycle can be executed which has it as a Parikh vector. By contrast, consider the right-hand side of Figure 28. The reachability graph also has two cycles, $(ac)^*$ and $(bc)^*$. However, they are not transition-disjoint, because c belongs to both.

In essence, in a bounded, reversible, persistent Petri net, realisable minimal T-invariants describe 'independent' repetitive behaviours, and this observation can be extended: Suppose that $X_1, \ldots, X_n$ are as in the previous theorem. Then there are n bounded, persistent and reversible nets $N_1, \ldots, N_n$, such that each net N_i has exactly *one* minimal realisable T-invariant X_i and the reachability graph of N is isomorphic to the reachability graph of the place-disjoint union of the nets $N_1, \ldots, N_n$.

As a consequence, the case in which a persistent net has exactly one minimal realisable T-invariant X is of special interest and needs to be scrutinized. It may still be the case that (unlike in a connected T-system, cf. Theorem 5) such a T-invariant is not a multiple of 1. However, there are special conditions under which this is indeed the case, called *k-multiply marked nets*, or k-nets for short.

Let N be a net and let $k \geq 1$ be some positive integer number. For a marking M, the k-multiple marking $k{\cdot}M$ is defined by $(k{\cdot}M)(s) = k{\cdot}(M(s))$ for every place s. The net $k{\cdot}N$ is the same as the net N except that the initial marking $k{\cdot}M_0$ replaces the initial marking M_0 of N. The net $k{\cdot}N$ is called a *k-net*. It turns out that initial k-markings $k{\cdot}M_0$ have particularly pleasant properties (partly generalising those of Theorem 5) provided that $k \geq 2$.

Theorem 16. *Smallest cycles in $k{\cdot}N$ have Parikh vector* 1 *if $k \geq 2$*
Let $k \geq 2$ and let $(N, k{\cdot}M_0)$ be a plain, bounded, reversible and persistent k-net with exactly one minimal realisable T-invariant X. Then $X \leq 1$ and for any transition t, $X(t) = 0$ if and only if t is dead at $k{\cdot}M_0$. ■

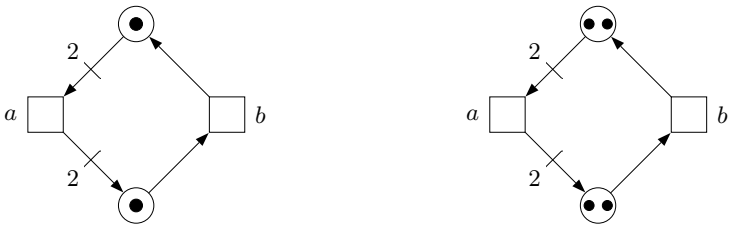


Fig. 32. A non-plain Petri net (l.h.s.) and its 2-multiple (r.h.s.) with minimal realisable T-invariant $(1\,2)^{\mathsf{T}}$

Plainness is important for Theorem 16 to hold. In Figure 32, all smallest cycles of the net on the right-hand side have Parikh vector $X = \mathcal{P}(abb)$, but $X > 1$, contrary to the conclusion of Theorem 16.

The statement made in Theorem 16 would not hold under the weaker assumption that N instead of $k{\cdot}N$ is persistent. For instance, let $k = 2$ and consider Figure 33. On the left-hand side, $X = (a \mapsto 1, b \mapsto 1, c \mapsto 2) = (1\,1\,2)^{\mathsf{T}}$ is the unique minimal realisable T-invariant, and it can be realised by the firing sequence $M_0[acbc\rangle M_0$. Note that $X \not\leq 1$. On the right-hand side, X is also the unique minimal realisable T-invariant, so that the conclusion of Theorem 16 is not true for this net. However, also one of the conditions of Theorem 16 is not satisfied, since the net is not persistent: executing a in the initial marking leads to a marking in which both a and b are enabled although their shared input place s carries only one token, hence producing a true conflict and destroying persistency. Thus, both requirements that $k{\cdot}N$ be persistent and that $k \geq 2$ are crucial for Theorem 16 to hold.

Next, we define an operation on transition sequences, called the *shuffle* or *arbitrary interleaving*. Intuitively, one may imagine some pack of cards to be divided into two halves and the second half be merged into the first. Instead of cards we may think of transitions, while the two half-packs correspond to sequences. Shuffling two sequences leaves the order of transitions stemming from

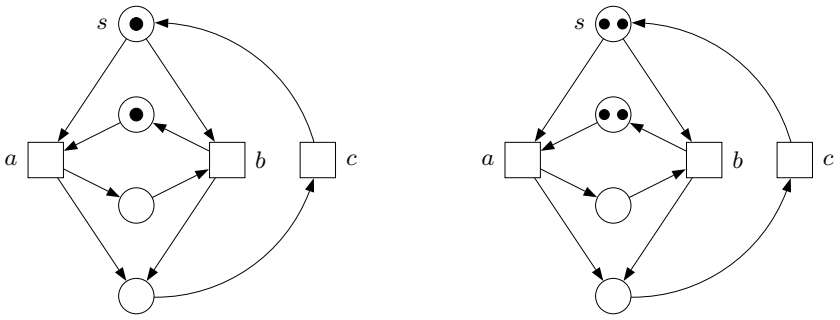


Fig. 33. A persistent Petri net (l.h.s.) and its non-persistent 2-multiple (r.h.s.)

one of the sequences unchanged. If two letters are not from the same sequence, however, we cannot predict their order in the resulting sequence.

Formally, the shuffle of two sequences v and w is a *set* of sequences written as $v \sqcup\!\sqcup w$. As an example, let $v = ab$ and $w = cbd$. Then

$$v \sqcup\!\sqcup w \;=\; \{abcbd,\ acbbd,\ acbdb,\ cabbd,\ cabdb,\ cbabd,\ cbadb,\ cbdab\}.$$

The shuffle can be extended canonically to sets of sequences. In general, it is associative and commutative.

Using shuffle, we define separability. This notion arises naturally in the context of k-nets.

Definition 25. *Weak and strong separability*
Let $k \geq 1$ and let $(N, k{\cdot}M)$ be any net with an initial k-marking $k{\cdot}M$.

A firing sequence $k{\cdot}M[\sigma\rangle$ is *weakly k-separable* from $k{\cdot}M$ (or just weakly separable if k and M are understood from the context) if there exist k sequences $\sigma_1, \ldots, \sigma_k$ such that

$$(\forall j, 1{\leq}j{\leq}k\colon M[\sigma_j\rangle \text{ in } (N, M)) \quad \text{and} \quad (\sum_{j=1}^{k} \mathcal{P}(\sigma_j)) = \mathcal{P}(\sigma). \tag{7}$$

A firing sequence $k{\cdot}M[\sigma\rangle$ is *strongly k-separable* from $k{\cdot}M$ if there exist k sequences $\sigma_1, \ldots, \sigma_k$ such that

$$(\forall j, 1{\leq}j{\leq}k\colon M[\sigma_j\rangle \text{ in } (N, M)) \quad \text{and} \quad \sigma \in \sigma_1 \sqcup\!\sqcup \ldots \sqcup\!\sqcup \sigma_k. \tag{8}$$

A k-net is weakly (strongly) separable if every sequence firable in its initial marking is weakly (strongly) separable from this k-marking. ■

Separability can be useful in verifying Petri nets. If some k-net $k{\cdot}N$ is separable, then it is sufficient to verify N rather than $k{\cdot}N$ because, as a rule, properties of the latter can easily be deduced from properties of the former. This can increase efficiency considerably if k is large, because the reachability graph of $k{\cdot}N$ could be much larger than that of N.

As an example, consider the two nets in Figure 34. On the left-hand side, a 2-marking is shown, where the set of tokens was split evenly into a hollow part and a solid part. The hollow tokens constitute M_0, and the solid tokens also constitute M_0. Thus, the whole marking is $2{\cdot}M_0$. Consider the firing sequence

$$2{\cdot}M_0\,[t_1t_2tt_1t_2\rangle.$$

This sequence can actually be fired in M_0, using only one of the two sorts of tokens, either just the hollow ones or just the solid ones. Hence $2{\cdot}M_0\,[\underbrace{t_1t_2tt_1t_2}_{\sigma}\rangle$ is strongly 2-separable by $M_0\,[\underbrace{t_1t_2tt_1t_2}_{\sigma_1}\rangle$ and $M_0\,[\underbrace{\varepsilon}_{\sigma_2}\rangle$. Consider the slightly longer firing sequence

$$2{\cdot}M_0\,[t_1t_2tt_1t_2t\rangle.$$

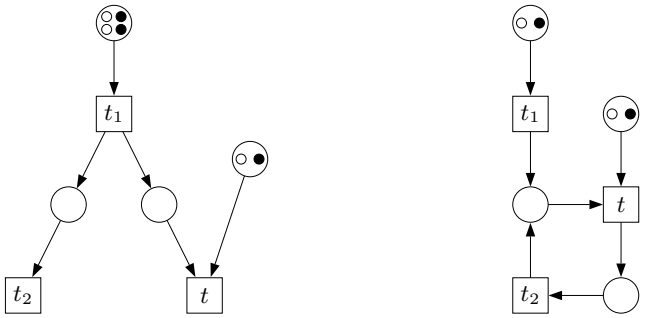


Fig. 34. Two 2-nets with hollow and solid tokens; separable (l.h.s.) and not separable (r.h.s.)

This sequence cannot be fired using only hollow tokens or only solid tokens, that is, we have $\neg\, M_0\,[t_1t_2tt_1t_2t\rangle$. If we try to 'prolong' the existing separation, we will fail, because t alone is not firable from M_0. Nevertheless, $2{\cdot}M_0\,[t_1t_2tt_1t_2t\rangle$ can be strongly 2-separated by $M_0\,[t_1t_2t\rangle$ and $M_0\,[t_1t_2t\rangle$, since $t_1t_2tt_1t_2t \in (t_1t_2t \sqcup\!\sqcup t_1t_2t)$. Intuitively, this corresponds to using hollow tokens for the first half and solid tokens for the second half of $t_1t_2tt_1t_2t$ (or the other way round).

Consider the net on the right-hand side of Figure 34. It again shows a 2-marking, and we have $2{\cdot}M_0\,[t_1tt_2t\rangle$. But no matter what kinds of tokens are used in order to fire t_1tt_2, the resulting marking activates t with mixed types of tokens, both a hollow one and a solid one. Formally, there are no two sequences σ_1 and σ_2 satisfying $M_0\,[\sigma_1\rangle$ and $M_0\,[\sigma_2\rangle$ and $t_1tt_2t \in (\sigma_1 \sqcup\!\sqcup \sigma_2)$. This shows that t_1tt_2t is not strongly 2-separable. There are not even two sequences σ_1 and σ_2 satisfying $M_0\,[\sigma_1\rangle$ and $M_0\,[\sigma_2\rangle$ and $\mathcal{P}(\sigma_1) + \mathcal{P}(\sigma_2) = \mathcal{P}(t_1tt_2t)$. Thus, t_1tt_2t is not even weakly separable.

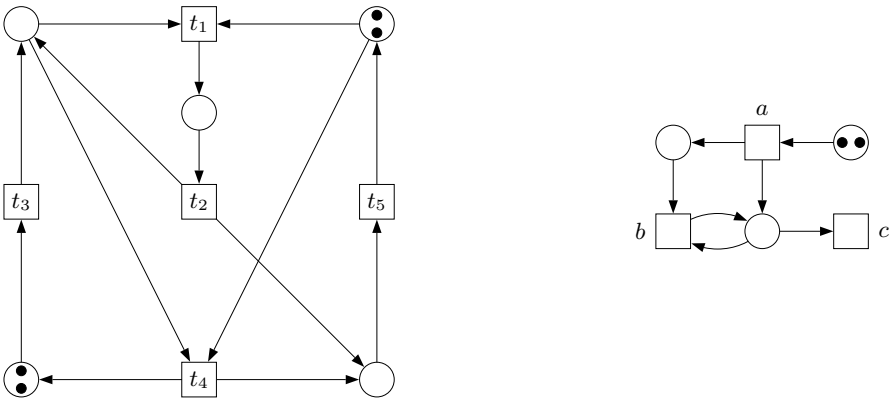


Fig. 35. Live, 2-bounded and not 2-separable (l.h.s.); weakly but not strongly separable (r.h.s.)

Figure 35(l.h.s.) depicts a 2-marking $2{\cdot}M$ which is live and bounded but not 2-separable. The reader is encouraged to find a non-separable firing sequence $2{\cdot}M\,[\sigma\rangle$. The 2-net shown on the right-hand side of Figure 35 is not strongly 2-separable from the indicated marking $2{\cdot}M$ since $2{\cdot}M[aacbbc\rangle$ cannot be obtained by shuffling two firing sequences from M. However, this 2-net is weakly 2-separable from $2{\cdot}M$. In particular, $\mathcal{P}(aacbbc) = \mathcal{P}(abc) + \mathcal{P}(abc)$, and clearly, $M[abc\rangle$. This 2-net is neither reversible nor persistent; e.g., $2{\cdot}M[acab\rangle$ and $2{\cdot}M[acac\rangle$ but $acacb$ cannot be fired from $2{\cdot}M$.

Separability cannot easily be checked directly on the reachability graph, because it is necessary to find a method for capturing all (possibly infinitely many) paths in the reachability graph of a k-marked net $(N, k{\cdot}M)$ and check them against k paths of k copies of the reachability graph of (N, M).

Nevertheless, both weak and strong separability can be deduced for persistent Petri nets under some further premises.

Theorem 17. *Weak and strong separability*
Let N be plain. Let $k \geq 1$ and let $k{\cdot}N$, with initial marking $k{\cdot}M_0$, be bounded, reversible, and persistent. If $k{\cdot}N$ has only one minimal realisable T-invariant, then $(N, k{\cdot}M_0)$ is weakly and strongly k-separable. ■

The proof of this result works roughly as follows. For $k = 1$, nothing has to be proved because every net is trivially 1-separable. For $k \geq 2$, Theorem 16 is exploited in an essential way. As a next step, weak separability is proved. Finally, in order to prove strong separability, the property of weak separability is used. All parts of this proof (except the case $k = 1$) are non-trivial.

Reversibility, plainness and persistency are important for Theorem 17 to hold. Figure 34 shows on the right-hand side a plain, bounded, non-reversible, persistent Petri net with a 2-marking $2{\cdot}M_0$ such that the firing sequence $2{\cdot}M_0[t_1tt_2t\rangle$ is not weakly 2-separable. The right-hand side of Figure 32 displays a non-plain, bounded, reversible, persistent 2-net with a 2-marking $2{\cdot}M_0$ in which the firing sequence $2{\cdot}M_0[a\rangle$ cannot be separated for obvious reasons. The net shown on the left-hand side of Figure 35 is not persistent but live, bounded, reversible and FC, showing that persistency cannot be omitted and that Theorem 17 does not hold for live and bounded FC-nets.

With the help of Theorem 15, Theorem 17 can be extended to bounded, reversible and persistent nets with several incomparable (mutually transition-disjoint) realisable T-invariants:

Theorem 18. *Strong separability for general bounded, reversible and persistent k-nets*
Let N be plain. Let $k \geq 1$ and let $k{\cdot}N$, with initial marking $k{\cdot}M_0$, be bounded, reversible, and persistent. Then $(N, k{\cdot}M_0)$ is weakly and strongly separable.
■

4.5 Bibliographical Remarks and Further Reading

The class of (place-)output-nonbranching nets is intimately related to system classes also known as *context-free processes* [CHS95] or *basic process algebra*

(BPA) [BW90]. The class of transition-input-nonbranching nets, also known as *communication-free nets* [Esp97], is intimately related to system classes otherwise known as *basic parallel processes* (BPP) [CHS93]. The class of conflict-free Petri nets has been introduced in [LR78] and studied, amongst others, in [HR89]. The class of BiCF nets has been studied in [GGS11].

Sections 4.2 and 4.3 are based on [LR78, Gra80, Yam81, HI92] and on [Kel75]. The class of persistent Petri nets has been studied from different perspectives and extended in various ways; see, e.g., [BO09]. A net with non-semilinear reachability graph can be found in [LR78]. In the context of workflow systems, (weak) separability was introduced in [HSV03] for Petri nets, as follows:

> For business applications, separability is important because it formalises the idea of independent cases... If we associate to each firing the consumption of some resource, like money or energy, then separability implies that the consumption of a batch of cases equals the sum of the individual consumptions.

In the area of security kernels, a related concept has been known for some time, cf. the seminal paper [Rus82]. The results quoted in section 4.4 are based on [BDW07, BD09, BD11]. Figure 35(l.h.s.) is due to Karsten Wolf.

Acknowledgements. The authors are grateful to two reviewers for their comments.

References

[BW90] Baeten, J.C.M., Weijland, W.P.: Process algebra. In: Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press (1990)

[BO09] Barylska, K., Ochmański, E.: Levels of Persistency in Place/Transition Systems. Fundamenta Informaticae 93, 33–43 (2009)

[BD09] Best, E., Darondeau, P.: A decomposition theorem for finite persistent transition systems. Acta Informatica 46, 237–254 (2009)

[BD11] Best, E.: P Darondeau: Separability in persistent Petri nets. Fundamenta Informaticae 112, 1–25 (2011)

[BDE92] Best, E., Desel, J., Esparza, J.: Traps characterize home states in free choice systems. Theoretical Computer Science 101, 161–176 (1992)

[BDW07] Best, E., Darondeau, P., Wimmel, H.: Making Petri nets safe and free of internal transitions. Fundamenta Informaticae 80, 75–90 (2007)

[BM85] Best, E., Merceron, A.: Frozen tokens and D-continuity: a study in relating system properties to process properties. In: Rozenberg, G. (ed.) APN 1984. LNCS, vol. 188, pp. 48–61. Springer, Heidelberg (1985)

[BV84] Best, E., Voss, K.: Free choice systems have home states. Acta Informatica 21, 89–100 (1984)

[CCS91] Campos, J., Chiola, G., Silva, M.: Properties and performance bounds for closed free choice synchronized monoclass queueing networks. IEEE Transactions on Automatic Control 36(12), 1368–1381 (1991)

[CHS93] Christensen, S., Hirshfeld, Y., Stirling, C.: Bisimulation equivalence is decidable for basic parallel processes. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 143–157. Springer, Heidelberg (1993)

[CHS95] Christensen, S., Hüttel, H., Stirling, C.: Bisimulation equivalence is decidable for all context-free processes. Information and Computation 121(2), 143–148 (1995)

[CHEP71] Commoner, F., Holt, A., Even, S., Pnueli, A.: Marked directed graphs. Journal of Computer and System Sciences 5, 511–523 (1971)

[Des92] Desel, J.: Struktur und Analyse von Free-Choice-Petrinetzen. In: Dissertation. Deutscher Universitätsverlag, Vieweg (1992)

[Des98] Desel, J.: Petrinetze, lineare Algebra und lineare Programmierung. Teubner-Texte zur Informatik, vol. 26. Band (1998)

[DE95] Desel, J., Esparza, J.: Free choice Petri nets. Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press (1995)

[Die10] Diestel, R.: Graph theory, `http://diestel-graph-theory.com/`

[Esp97] Esparza, J.: Petri nets, commutative context-free grammars, and basic parallel processes. Fundamenta Informaticae 31, 13–26 (1997)

[Esp98] Esparza, J.: Reachability in live and safe free-choice Petri nets is NP-complete. Theoretical Computer Science 198(1-2), 211–224 (1998)

[Far1902] Farkas, J.: Über die Theorie der einfachen Ungleichungen. Journal für die Reine und Angewandte Mathematik 124, 1–24 (1902)

[GHM03] Gaujal, B., Haar, S., Mairesse, J.: Blocking a transition in a free choice net and what it tells about its throughput. Journal of Computer and System Sciences 66, 515–548 (2003)

[GL73] Genrich, H.J., Lautenbach, K.: Synchronisationsgraphen. Acta Informatica 2, 143–161 (1973)

[GT84] Genrich, H.J., Thiagarajan, P.S.: A theory of bipolar synchronization schemes. Theoretical Computer Science 30, 241–318 (1984)

[GGS11] van Glabbeek, R.J., Goltz, U., Schicke, J.-W.: On causal semantics of Petri nets. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 43–59. Springer, Heidelberg (2011)

[Gor1873] Gordan, P.: Über die Auflösung linearer Gleichungen mit reellen Coefficienten. Mathematische Annalen 6, 23–28 (1873)

[Gra80] Grabowski, J.: The decidability of persistence for vector addition systems. Information Processing Letters 11, 20–23 (1980)

[HMW10] Habermehl, P., Meyer, R., Wimmel, H.: The downward-closure of Petri net languages. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 466–477. Springer, Heidelberg (2010)

[Hac72] Hack, M.H.T.: Analysis of production schemata by Petri nets. TR–94, MIT–MAC (1972); Corrections (1974)

[Hac74] Hack, M.H.T.: Decision problems for Petri nets and vector addition systems. Computation Structures Memo 95, Project MAC, MIT (1974)

[HSV03] van Hee, K., Sidorova, N., Voorhoeve, M.: Soundness and separability of workflow nets in the stepwise refinement approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003)

[HI92] Hiraishi, K., Ichikawa, A.: On structural conditions for weak persistency and semilinearity of Petri nets. Theoretical Computer Science 93(2), 185–199 (1992)

[HR89] Howell, R.R., Rosier, L.E.: Problems concerning fairness and temporal logic for conflict-free Petri nets. Theoretical Computer Science 64(3), 305–329 (1989)

[KM69] Karp, R.M., Miller, R.E.: Parallel program schemata. Journal of Computer and System Sciences 3, 147–195 (1969)

[Kel75] Keller, R.M.: A fundamental theorem of asynchronous parallel computation. In: Tse-Yun, F. (ed.) Parallel Processing. LNCS, vol. 24, pp. 102–112. Springer, Heidelberg (1975)

[Kos82] Kosaraju, S.R.: Decidability of reachability in vector addition systems. In: Proceedings of the 14th Annual ACM STOC, pp. 267–281 (1982)

[Lam92] Lambert, J.L.: A structure to decide reachability in Petri nets. Theoretical Computer Science 99, 79–104 (1992)

[Lam77] Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering 3(2), 125–143 (1977)

[Lam] Lamport, L.: Personal communication (around 1995), `http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html` entry 23 (Last accessed April 14, 2012)

[LR78] Landweber, L.H., Robertson, E.L.: Properties of conflict-free and persistent Petri nets. Journal of the ACM 25(3), 352–364 (1978)

[Lau73] Lautenbach, K.: Exakte Bedingungen der Lebendigkeit für eine Klasse von Petri-Netzen. GMD, Bericht Nr. 82 (Dissertation) (1973)

[LR94] Lautenbach, K., Ridder, H.: Liveness in bounded Petri nets which are covered by T-invariants. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 358–375. Springer, Heidelberg (1994)

[Ler09] Leroux, J.: The general vector addition system reachability problem by Presburger inductive invariants. In: 24th IEEE Symposium on Logic in Computer Science (LICS 2009), Los Angeles, California, USA, August 11-14, pp. 4–13 (2009)

[MM98] Matsumoto, T., Mayano, Y.: Reachability criterion for Petri nets with known firing vectors. IEICE Trans. Fundamentals E81-A(4), 628–634 (1998)

[May80] Mayr, E.W.: Ein Algorithmus für das allgemeine Erreichbarkeitsproblem bei Petrinetzen und damit zusammenhängende Probleme. Technischer Bericht der Technischen Universität München TUM-I8010 (Dissertation). Institut für Informatik (1980)

[May84] Mayr, E.W.: An algorithm for the general Petri net reachability problem. SIAM Journal of Computing 13(3), 441–460 (1984)

[Mur89] Murata, T.: Petri nets: properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)

[Pet81] Peterson, J.L.: Petri net theory and the modeling of systems. Prentice-Hall (1981)

[Pet62] Petri, C.A.: Kommunikation mit Automaten. Dissertation, Institut für Instrumentelle Mathematik, Bonn, Schriften des IMM Nr. 2 (1962)

[PW03] Priese, L., Wimmel, H.: Petri-Netze, 376 pages. Springer (2003) ISBN: 3-540-44289-8

[RTS98] Recalde, L., Teruel, E., Silva, M.: On linear algebraic techniques for liveness analysis of P/T Systems. Journal of Circuits, Systems, and Computers 8(1), 223–265 (1998)

[Rei85] Reisig, W.: Petri nets. EATCS Monographs on Theoretical Computer Science, vol. 4. Springer (1985)

[Rus82] Rushby, J.: Proof of Separability – a Verification Technique for a Class of Security Kernels. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 352–367. Springer, Heidelberg (1982)

[Schr86] Schrijver, A.: Theory of linear and integer programming. Wiley (1986)
[TCCS92] Teruel, E., Chrząstowski-Wachtel, P., Colom, J.M., Silva, M.: On weighted T-systems. In: Jensen, K. (ed.) ICATPN 1992. LNCS, vol. 616, Springer, Heidelberg (1992)
[TS96] Teruel, E., Silva, M.: Structure theory of equal conflict systems. Theoretical Computer Science 153(1-2), 271–300 (1996); Special volume on Petri nets
[Weh09] Wehler, J.: Free-choice Petri nets without frozen tokens, and bipolar synchronization systems. Fundamenta Informaticae 96, 1–38 (2009)
[Weh10] Wehler, J.: Simplified proof of the blocking theorem for free-choice Petri nets. Journal of Computer and System Sciences 76, 532–537 (2010)
[Wim04] Wimmel, H.: Infinity of intermediate states is decidable for petri nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 426–434. Springer, Heidelberg (2004)
[Yam81] Yamasaki, H.: On weak persistency of Petri nets. Information Processing Letters 13(3), 94–97 (1981)
[YHM01] Yamasaki, H., Huang, J.S., Murata, T.: Reachability analysis of Petri nets via structural and behavioral classifications of transitions. Petri Net Newsletter 60, 4–21 (2001); Gesellschaft für Informatik
[YY03] Yen, H., Yu, L.: Petri nets with simple circuits. In: Warnow, T.J., Zhu, B. (eds.) COCOON 2003. LNCS, vol. 2697, pp. 149–158. Springer, Heidelberg (2003)

Causality in Extensions of Petri Nets

Jetty Kleijn[1] and Maciej Koutny[2]

[1] LIACS, Leiden University
Leiden, 2300 RA, The Netherlands
kleijn@liacs.nl
[2] School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU, United Kingdom
maciej.koutny@ncl.ac.uk

Abstract. The causal semantics of standard net classes like Elementary Net Systems and Place/Transition Nets, is typically expressed in terms of partially ordered sets of transition occurrences. In each such partial order, causally related occurrences are ordered while concurrent transition occurrences remain unordered. Partial order semantics can, in particular, support model checking by efficient verification techniques based on net unfoldings.

To enhance the modelling power of standard net classes, one can introduce different forms of 'testing' using, for example, inhibitor arcs. However, the causal semantics of such extended nets can often no longer be described solely in terms of partial orders. In this paper, we explain what modifications to the partial order semantics are needed in order to provide a satisfactory treatment for nets with activator, inhibitor and mutex arcs. On the technical side, the proposed solution is based on causal structures which enrich partial orders with additional order relations corresponding to other aspects of causality. With EN-systems as our starting point, we discuss how their extensions can be treated using these richer notions of causality.

Keywords: elementary net systems, activator arcs, inhibitor arcs, mutex arcs, semantical framework, step sequences, processes, causality semantics.

1 Introduction

In order to be able to verify complex, distributed systems, i.e., to guarantee correctness of their behaviour, one has to understand the relations between concurrently ongoing operations. This involves, in particular, providing appropriate mathematical abstractions to capture the operational properties of such systems.

Petri nets are a system model related to state machines and similar, sequential, behaviour defining devices. However, the states of Petri nets are distributed (over so-called places) and also their actions (transitions, in Petri net terms) occur purely locally. Whether or not a transition can occur, depends only those components (places) of the state to which it is directly related. Moreover, when it occurs, it affects only neighbouring places. Hence, each transition occurrence (an event) leads to a local change of state. All this induces local interactions between transition occurrences making it possible to extract from a run of a Petri net, the essential causal relationships between events.

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 225–254, 2013.

These local interactions can be derived from so-called processes, i.e., labelled acyclic nets representing the unfolding of a net corresponding to a single execution (with all choices and conflicts resolved). Abstracting from the places leads to a causal semantics expressed in terms of partially ordered sets of occurrences of transitions: causally related events are ordered, while concurrent events remain unordered. Each such partial order describes the causal structure of a single concurrent history or run of the system and as such represents several — closely related — (step) sequences of (simultaneously occurring) transitions, each of them being a possible observation of that run. The standard net classes of Elementary Net Systems (or EN-systems) and Place/Transition Nets (or PT-nets) are typical examples of this approach [1,27].

As an example, consider Figure 1(a) depicting an EN-system with three step sequences involving the executions of transitions a, b and c, viz. $\sigma_1 = \{a, b\}\{c\}$, $\sigma_2 = \{a\}\{b\}\{c\}$ and $\sigma_3 = \{b\}\{a\}\{c\}$. They can be seen as observations of a single history underpinned by a causal partial order in which a and b are unordered and both a and b precede c.

Consistency between the different levels of abstraction at which one captures the concurrency in the behaviour can be established within a generic approach (the *semantical framework* of [19]) aimed at fitting together systems (i.e., nets from a certain class of Petri nets), abstract causal orders and individual observations.

Partial order semantics as just described can support efficient verification techniques. Rather than exploring the full state space of a system constructed from sequential observations, one uses unfoldings, see [4] for a general description of this approach. The idea behind the resulting more efficient algorithms is to exploit the concurrency (unorderedness) in the behaviour to alleviate the state space explosion problem. For Petri nets, unfoldings and nonsequential net processes provide a truly concurrent semantics with partial orders as a succinct representation of related observations. Unfoldings based on the branching processes from [3] in which also all choices are modelled, are the basis for efficient verification algorithms [5,18,23].

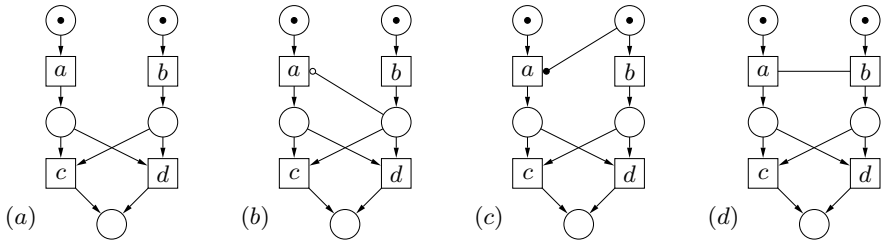


Fig. 1. An EN-system (a); an EN-system with an inhibitor arc joining the output place of transition b with transition a implying that a cannot be fired if the output place of b is not empty (b); an EN-system with an activator arc joining the input place of transition b with transition a implying that a can be fired provided that the input place of b is not empty (c); and an EN-system with a mutex arc between transitions a and b implying that the two transitions cannot be fired in the same step (d)

To enhance the modelling power of the standard net classes one can introduce different forms of 'testing', for example, testing for the absence of a token using inhibitor arcs. This may imply that the causal semantics of such extended Petri net models can no longer be described solely in terms of partial orders.

Figure 1(b) depicts an EN-system with an inhibitor arc. Such an arc between a place and a transition indicates that the place has to be empty for the transition to be able to fire. Hence this net has only two step sequences involving transitions a, b and c, namely $\sigma_1 = \{a, b\}\{c\}$ and $\sigma_2 = \{a\}\{b\}\{c\}$. This is because a can occur before b or simultaneously with b but 'not later than' b (weak causality). These two step sequences can be seen as belonging to the abstract causal history underpinned *not* by causal partial orders but rather by causality structures introduced in [14] — called *stratified order structures* — based on causal partial orders and, in addition, weak causal partial orders. Another form of testing is portrayed by the net in Figure 1(c) which depicts an EN-system with an activator arc. Activator arcs (see, e.g., [24]) are closely related to inhibitor arcs. Such a 'testing' arc between a place and a transition means that the place has to be non-empty for the transition to be able to fire. As a result, both step sequences and abstract causal histories of this net are exactly the same as in the previous example.

Yet another example, in Figure 1(d), depicts an EN-system with a mutex arc. Such an arc means that the two connected transitions may occur in any order but not simultaneously (commutativity). Hence this net has two step sequences involving transitions a, b and c, namely $\sigma_2 = \{a\}\{b\}\{c\}$ and $\sigma_3 = \{b\}\{a\}\{c\}$. They belong to an abstract history underpinned by causality structures introduced in [7,10] — called *generalised stratified order structures* — based on causal partial orders together with weak causal partial orders and, in addition, a commutativity relation which tells what pairs of events cannot belong to the same step.

In this paper, we explain what modifications to the partial order semantics are needed in order to provide a satisfactory treatment for nets with inhibitor, activator and mutex arcs. The model which we extend with these new types of arcs are Elementary Net systems [27]. This model is *the* basic class of Petri nets and is particularly suited for the study of fundamental properties of concurrent systems. In particular, EN-systems are the typical concurrency model in which event independence, simultaneity, and unorderedness amount to basically the same semantical phenomenon, making partial orders exactly the right abstract model for their behaviour. We will discuss how the extended classes of EN-systems can be treated with the richer notions of causal semantics using the generic approach provided by the semantical framework of [19]. Finally, we will bring Place/Transition Nets into our discussion and reflect upon similarities and differences with the EN-systems approach. As a tutorial survey, this paper provides no proofs, but rather provides 'facts' with references for proofs and more background information, given per (sub)section.

2 Preliminaries

Composing two functions $f : X \to 2^Y$ and $g : Y \to 2^Z$ is defined by $g \circ f(x) = \bigcup_{y \in f(x)} g(y)$, for all $x \in X$. Restricting function f to a subset Z of X is denoted by $f|_Z$. Similarly, the restriction of a binary relation $R \subseteq X \times Y$ to a subset Z of $X \times Y$

is denoted by $R|_Z$. We may use the infix notation $x\,R\,y$ to denote that $(x,y) \in R$. The composition $R \circ Q$ of two relations $R \subseteq X \times Y$ and $Q \subseteq Y \times Z$ comprises all pairs (x,z) in $X \times Z$ for which there is y in Y such that $(x,y) \in R$ and $(y,z) \in Q$. We assume the following notions and notations:

- $R^{-1} = \{(y,x) \mid (x,y) \in R\}$. (reverse)
- $R^0 = id_X = \{(x,x) \mid x \in X\}$. (identity)
- $R^n = R^{n-1} \circ R$. (n-th power, $n \geq 1$).
- $R^+ = R^1 \cup R^2 \cup \ldots$. (transitive closure)
- $R^* = R^0 \cup R^+$. (reflexive transitive closure)
- $R^{\mathsf{sym}} = R^0 \cup R^{-1}$. (symmetric closure)
- R is symmetric, reflexive, irreflexive, transitive if, respectively, $R = R^{-1}$, $id_X \subseteq R$, $id_X \cap R = \varnothing$, $R \circ R \subseteq R$.
- R is acyclic if R^+ is irreflexive.

A *relational structure* is a tuple $rs = (X, Q_1, \ldots, Q_n)$ where X is a finite set called *domain*, and the Q_i's are binary relations on X (we can select components using the subscript rs, e.g., X_{rs}). For relational structures with the same domain and arity, rs and rs', we write $rs \subseteq rs'$ if the subset inclusion holds component-wise. The intersection $\bigcap \mathcal{R}$ of a set $\mathcal{R}$ of relational structures with the same arity and domain is defined component-wise.

A *sequence* over a finite set X is a finite string $x_1 \ldots x_n$ of symbols x_i from X. A *step* over X is a non-empty subset of X, and a *step sequence* over X is a finite string $X_1 \ldots X_n$ of steps. A step sequence is *singular* if the X_i's are mutually disjoint. The empty (step) sequence, corresponding to the case $n = 0$, is denoted by λ. As singleton sets can be identified with their only elements, sequences can be regarded as special step sequences. Moreover, the set brackets of singleton sets will be omitted.

A *labelling* ℓ of a set X is a function from X to a set of labels $\ell(X)$, and a *labelled set* is a pair (X, ℓ) where X is a set and ℓ is a labelling of X. The labelling is extended to finite sequences of elements x_i of X by $\ell(x_1 \ldots x_n) = \ell(x_1) \ldots \ell(x_n)$, and to finite sequences of subsets X_i of X by $\ell(X_1 \ldots X_n) = \ell(X_1) \ldots \ell(X_n)$. To make the labelling explicit, we will sometimes denote a labelled step sequence by (σ, ℓ). We will also use $\phi(\sigma, \ell) = \ell(\sigma)$ when we want to 'forget' about the underlying elements but rather focus on the step sequence $\ell(\sigma)$ over $\ell(X)$.

> We *assume* throughout that all sets in this paper are *labelled sets*, with the default labelling simply being the identity function. If the actual labelling is irrelevant for a particular definition or result, it may be omitted. Moreover, whenever it is stated that two domains are the same, we implicitly assume that their labellings are identical.

3 Causal Partial Orders and Order Structures

To capture the intrinsic causal relationships between events occurring in a concurrent system history, one normally resorts to using a suitable *ordering relation*. In its basic form, such a relation is a partial order (reflecting the generally accepted view that

causality is transitive and acyclic). However, for systems with a complex structure, partial orders may need to be extended to more expressive *order structures* which support additional relations between events, such as *weak* causality. We will present two kinds of such extended order structures.

When using (causal) ordering relations in the treatment of concurrent histories, there are two crucial issues which need to be satisfactorily addressed. The first is the relationship with their associated executions or observations, typically captured by sequences or step sequences of events. To be meaningful, an ordering relation should be a faithful abstraction of a set of executions in the sense that each of these corresponds to the given order (should be allowed as an execution). Moreover, there should be an unambiguous way of deriving an ordering relation from a set of observations, by capturing all essential causal orderings between events while ignoring coincidental ordering in any concrete observation. We will refer to such a property as *Abstraction*. The second issue is related to the way ordering relations are derived. Intuitively, an overall causal ordering relation should be built up from smaller, more direct local, causal ordering relations by applying some notion of transitivity. We will refer to such an operation as *Closure*.

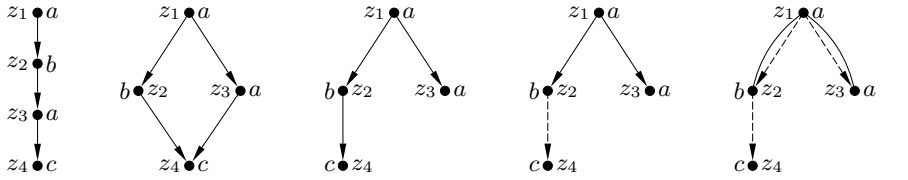
3.1 Partial Orders

A *partially ordered set* (or poset) $po = (X, \prec)$ is a relational structure comprising a finite set X and an irreflexive and transitive binary relation $\prec$ on X. Two distinct elements x, y of X are *unordered*, $x \frown y$, if neither $x \prec y$ nor $y \prec x$. We denote $a \widehat{\prec} b$ if $a \prec b$ or $a \frown b$.

Intersecting posets to filter out their common ordering is a sound operation yielding a new poset.

Fact 1 (poset intersection). *If $\mathcal{PO}$ is a non-empty set of posets with a common domain, then $\bigcap \mathcal{PO}$ is a poset with the same domain.*

A poset po is *total* (or linear) if all pairs of distinct elements of X are ordered, and *stratified* (or weak) if $\frown \cup id_X$ is an equivalence relation. Note that all total posets are also stratified. If a poset represents a history of a concurrent system, then $x \prec y$ means that x can only be observed before y, while $x \frown y$ means that x and y can be observed in any order, even simultaneously. In Figure 2, tpo_0 is a total poset and spo_0 is a stratified poset.



tpo_0: total spo_0: stratified po_0: neither total nor stratified sos_0: SO-structure $gsos_0$: GSO-structure

Fig. 2. Hasse diagrams of posets and order structures showing also the labels (a, b and c) of their elements. Solid arcs represent $\prec$, dashed arcs represent $\sqsubset$, and solid edges represent $\rightleftharpoons$.

To formulate the *Abstraction* property for posets, we first need to make it clear which executions correspond to a given (causal) poset po. A total poset tpo is a *linearisation* of po if $po \subseteq tpo$, while a stratified poset spo is a *stratification* of po if $po \subseteq spo$. (That is, po is a faithful abstraction of tpo and spo.) We denote this respectively by $tpo \in \mathsf{lin}(po)$ and $spo \in \mathsf{strat}(po)$. In Figure 2, $tpo_0 \in \mathsf{lin}(po_0)$ and $spo_0 \in \mathsf{strat}(po_0)$. Conversely, po captures all essential orderings present in its linearisations or stratifications, respectively.

Fact 2 (poset abstraction [28]). *For every poset po, $\mathsf{lin}(po) \neq \varnothing$ and*

$$po = \bigcap \mathsf{lin}(po) \ .$$

The above fact, known as *Szpilrajn's Theorem*, implies that a poset is uniquely determined by the intersection of its linearisations. The same holds for stratifications.

Fact 3 (poset abstraction [15]). *For every poset po, $\mathsf{strat}(po) \neq \varnothing$ and*

$$po = \bigcap \mathsf{strat}(po) \ .$$

The *Poset Closure* property described next is simple and indeed standard, but it is still a good idea to state it explicitly as we will soon generalise it to more complicated order structures.

A *pre-poset* is a relational structure $\varrho = (X, \prec)$ such that $\prec^+$ is irreflexive. In such a case, its *po-closure* is defined as $\varrho^{\mathsf{po}} = (X, \prec^+)$. Intuitively, $\prec$ indicates which of the executed actions are *directly* causally related and ϱ^{po} provides a full account of both direct and indirect (derived) causality between events. Therefore, we require that $\prec$ be acyclic, i.e., $\prec^+$ is irreflexive. Then its transitive closure yields the overall causality relationship.

Fact 4 (poset closure). *For every pre-poset ϱ, ϱ^{po} is a poset.*

As already mentioned, individual executions of a concurrent system are often represented by sequences of events or sequences of sets of simultaneously occurring events (step sequences). Both are language theoretic rather than order theoretic notions, but there is a straightforward way to move between these two representations. Given a stratified poset $spo = (X, \prec)$, there is a unique enumeration $X_1, \ldots, X_k$ of the equivalence classes of the relation $\nsucc \cup id_X$ such that $x \prec y$, for all $x \in X_i$ and $y \in X_j$ and $i < j$. We then associate with spo the singular step sequence $\mathsf{steps}(spo) = X_1 \ldots X_k$. Conversely, if $\sigma = X_1 \ldots X_k$ ($k \geq 0$) is a singular step sequence, then

$$\mathsf{spo}(\sigma) = (\bigcup_i X_i, \bigcup_{i<j} X_i \times X_j)$$

is the stratified poset associated with σ. In Figure 2,

$$\begin{array}{ll} \mathsf{steps}(tpo_0) = z_1 z_2 z_3 z_4 & \mathsf{spo}(z_1 z_2 z_3 z_4) = tpo_0 \\ \mathsf{steps}(spo_0) = z_1 \{z_2, z_3\} z_4 & \mathsf{spo}(z_1 \{z_2, z_3\} z_4) = spo_0 \ . \end{array}$$

Fact 5 (posets and step sequences). *$spo = \mathsf{spo}(\mathsf{steps}(spo))$, for every stratified poset spo, and $\sigma = \mathsf{steps}(\mathsf{spo}(\sigma))$, for every singular step sequence σ.*

Hence we can *identify* each stratified poset spo with $\mathsf{steps}(spo)$ or, equivalently, *identify* each singular step sequence σ with $\mathsf{spo}(\sigma)$. This also applies to labelled stratified posets and labelled singular step sequences.

3.2 Stratified Order Structures

Posets capture an 'earlier than' relationship between the elements of their domains. Their first extension we consider consists in introducing the concept of a weaker — 'not later than' — relationship.

A *stratified order structure* (or SO-structure) $sos = (X, \prec, \sqsubset)$ comprises two binary relations, $\prec$ (*causality*) and $\sqsubset$ (*weak causality*, in diagrams represented by dashed arcs, see Figure 2) on a finite set X such that, for all $x, y, z \in X$:

$$S1: \ x \not\sqsubset x \qquad S3: \ x \sqsubset y \sqsubset z \ \wedge \ x \neq z \implies x \sqsubset z$$
$$S2: \ x \prec y \implies x \sqsubset y \qquad S4: \ x \sqsubset y \prec z \ \vee \ x \prec y \sqsubset z \implies x \prec z \ .$$

Intuitively, $\prec$ represents the 'earlier than' relationship in X, and $\sqsubset$ the 'not later than' relationship. Note that $\prec$ is a partial order, and $x \prec y$ implies $y \not\sqsubset x$. It is easily seen that if spo is a stratified poset, then the relational structure defined by $\mathsf{sos}(spo) = (X_{spo}, \prec_{spo}, \sqsupset_{spo})$ is an SO-structure.

Again, intersecting SO-structures to filter out their common orderings is a sound operation yielding a new SO-structure.

Fact 6 (sos intersection). *If $\mathcal{SOS}$ is a non-empty set of* SO*-structures with a common domain, then $\bigcap \mathcal{SOS}$ is an* SO*-structure with the same domain.*

To formulate the *Abstraction* property for SO-structures, we first need to define executions corresponding to a given SO-structure sos. A stratified poset spo is an *extension* of sos if $sos \subseteq \mathsf{sos}(spo)$. (Thus sos is a faithful abstraction of spo.) We denote this by $spo \in \mathsf{ext}(sos)$. In Figure 2, $tpo_0, spo_0 \in \mathsf{ext}(sos_0)$.

Fact 7 (sos abstraction). *For every* SO*-structure sos, $\mathsf{ext}(sos) \neq \varnothing$ and*

$$sos = \bigcap \mathsf{sos}(\mathsf{ext}(sos)) \ .$$

The *Closure* property for SO-structures generalises the notion of po-closure introduced for posets. A *pre-*SO*-structure* is a relational structure $\varrho = (X, \prec, \sqsubset)$ such that the relation $\gamma \circ \prec \circ \gamma$ is irreflexive, where $\gamma = (\prec \cup \sqsubset)^*$. Then its *so-closure* is:

$$\varrho^{\mathsf{so}} = (X, \gamma \circ \prec \circ \gamma, \gamma \setminus id_X) \ .$$

Note that in a pre-SO-structure ϱ there are no $x_0, x_1, \ldots, x_n = x_0$ such that $x_0 \prec x_1$ and, for all $0 < i < n$, $x_i \prec x_{i+1}$ or $x_i \sqsubset x_{i+1}$. This can be regarded as a counterpart of the acyclicity required of pre-posets.

Fact 8 (sos closure). *For every pre-*SO*-structure ϱ, ϱ^{so} is an* SO*-structure.*

Stratified order structures were independently introduced in [6] and [12]. Their theory has been presented in [15], and they have been used, for example, to model inhibitor and priority systems, asynchronous races and synthesis problems (see, e.g., [17]).

3.3 Generalised Stratified Order Structures

The second extension of causal posets introduces a representation of 'non-simultaneity'.

A *generalised* SO-*structure* (or GSO-structure) $gsos = (X, \rightleftharpoons, \sqsubset)$ comprises two irreflexive relations, $\rightleftharpoons$ (*commutativity*, which is symmetric) and $\sqsubset$ (*weak causality*, as before) on X such that $(X, \rightleftharpoons \cap \sqsubset, \sqsubset)$ is an SO-structure. Note that commutativity represents the 'earlier than or later than, but never simultaneous' relationship. Accordingly, $\rightleftharpoons \cap \sqsubset$ represents the 'earlier than' relationship, and so it is required that together with $\sqsubset$ it forms an SO-structure. In fact, one could have defined GSO-structures as $gsos = (X, \prec, \sqsubset, \rightleftharpoons)$ making them a direct generalisation of SO-structures. However, it is always the case that $\prec$ is the same as the intersection of $\sqsubset$ and $\rightleftharpoons$, and so it can be omitted. It is easily seen that if spo is a stratified poset, then the relational structure $\mathsf{gsos}(spo) = (X_{spo}, \prec^{\mathsf{sym}}_{spo}, \rightleftharpoons_{spo})$ is a GSO-structure.

Also in this case, intersecting GSO-structures to filter out their common orderings is a sound operation yielding a new GSO-structure.

Fact 9 (gsos intersection). *If $\mathcal{GSOS}$ is a non-empty set of* GSO-*structures with a common domain, then $\bigcap \mathcal{GSOS}$ is an* GSO-*structure with the same domain.*

To formulate the *Abstraction* property for GSO-structures, we need to define which executions would correspond to a given GSO-structure $gsos$. A stratified poset spo is an *extension* of $gsos$ if $gsos \subseteq \mathsf{gsos}(spo)$. (Thus $gsos$ is a faithful abstraction of spo.) We denote this by $spo \in \mathsf{ext}(gsos)$. In Figure 2, $spo_0 \in \mathsf{ext}(gsos_0)$. We then obtain that GSO-structures are fully determined by their extensions.

Fact 10 (gsos abstraction). *For every* GSO-*structure $gsos$, $\mathsf{ext}(gsos) \neq \varnothing$ and*

$$gsos = \bigcap \mathsf{gsos}(\mathsf{ext}(gsos)) \ .$$

The *Closure* property for GSO-structures generalises the notion of so-closure introduced for SO-structures. A *pre*-GSO-*structure* is a relational structure $\varrho = (X, \prec, \sqsubset, \rightleftharpoons)$ based on local relationships between events such that the relation $\alpha^{\mathsf{sym}} \cup \beta^{\mathsf{sym}} \cup \rightleftharpoons$ is irreflexive and symmetric, where

$$\alpha = \gamma \circ \prec \circ \gamma \quad \text{and} \quad \beta = \sqsubset^* \circ (\rightleftharpoons \cap \sqsubset^*) \circ \sqsubset^* \quad \text{and} \quad \gamma = (\prec \cup \sqsubset)^* \ .$$

In such a case, its *gso-closure* is defined as $\varrho^{\mathsf{gso}} = (X, \alpha^{\mathsf{sym}} \cup \beta^{\mathsf{sym}} \cup \rightleftharpoons, \gamma \setminus id_X)$. Note that $\rightleftharpoons$ relates events that cannot be executed simultaneously.

Fact 11 (gsos closure). *For every pre-*GSO-*structure ϱ, ϱ^{gso} is a* GSO-*structure.*

Generalised SO-structures were introduced in [7] to represent the most general concurrent histories in the approach of [13]. They were investigated in [10], and used to provide nets comprising mutex arcs with a causal semantics in [21].

4 Elementary Net Systems

All net models considered in this paper have a net as their underlying structure.

A *net* $N = (P, T, F)$ comprises disjoint finite sets of nodes, P and T, called respectively *places* and *transitions*, and the *flow* relation $F \subseteq (T \times P) \cup (P \times T)$. A *marking* of N is a set of places. In diagrams, places (local states) are represented by circles, transitions (actions) by rectangles, the flow relation by directed arcs, and a marking (global state) by *tokens* (small black dots) drawn inside places. The *inputs* and *outputs* of a node x are the sets ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$; moreover, ${}^\bullet x^\bullet = {}^\bullet x \cup x^\bullet$. It is assumed that ${}^\bullet t \neq \varnothing \neq t^\bullet$, for every transition t. The dot-notation extends to sets X of nodes in the usual way, e.g., ${}^\bullet X = \bigcup\{{}^\bullet x \mid x \in X\}$.

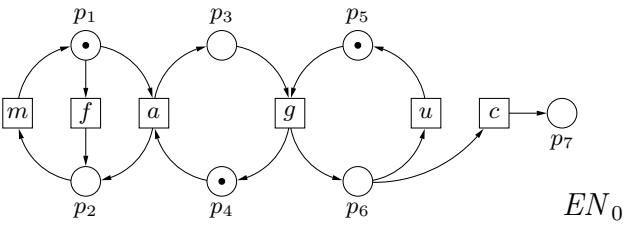


Fig. 3. EN-system model of a producer/consumer system

Figure 3 shows a net model of a system consisting of a producer, a buffer of capacity one, and a consumer. The producer can execute: m (making an item), a (adding a new item to the buffer), and f (failing to add an item). The consumer can execute: g (getting an item), u (using the item), and c (completing the work). The buffer executes cyclically the a and g actions. The three components operate independently with shared actions being executed jointly. Figure 3 also shows an (initial) marking $M = \{p_1, p_4, p_5\}$.

Net executions can be captured by sequences of steps of transitions. A *step* of a net is a set U of transitions such that ${}^\bullet t^\bullet \cap {}^\bullet v^\bullet = \varnothing$, for all $t \neq v \in U$. It is *enabled* at a marking M if ${}^\bullet U \subseteq M$ and $U^\bullet \cap M = \varnothing$. In such a case, the *execution* of U leads to marking $M' = (M \backslash {}^\bullet U) \cup U^\bullet$. We denote this by $M[U\rangle M'$.

A *step sequence* from a marking M to a marking M' is a sequence $\sigma = U_1 \ldots U_n$ $(n \geq 0)$ of non-empty steps U_i such that $M[U_1\rangle M_1, \ldots, M_{n-1}[U_n\rangle M'$, for some $M_1, \ldots, M_{n-1}$. We denote this by $M[\sigma\rangle M'$, and call M' *reachable* from M. When all steps U_i are singletons, σ is a *firing sequence*. For the net in Figure 3, we have:

$$\{p_2, p_3, p_6\}\ [m\rangle\ \{p_1, p_3, p_6\} \qquad \{p_1, p_4, p_5\}\ [a\{m, g\}\{a, c\}m\rangle\ \{p_1, p_3, p_7\}$$
$$\{p_2, p_3, p_6\}\ [\{m, c\}\rangle\ \{p_1, p_3, p_7\} \qquad \{p_1, p_4, p_5\}\ [amgacm\rangle\ \{p_1, p_3, p_7\}\ .$$

An EN-*system* is a tuple $EN = (P, T, F, M_{init})$ such that (P, T, F) is its *underlying* net, and M_{init} is an *initial* marking. Moreover, $\mathsf{steps}(EN)$ and $\mathsf{fseq}(EN)$ comprise respectively all the step sequences and all firing sequences from the initial marking M_{init}. Figure 3 depicts an EN-system with $\mathsf{steps}(EN_0) = \{\lambda, a, ag, am, a\{g, m\}, \ldots\}$ and $\mathsf{fseq}(EN_0) = \{\lambda, a, ag, am, agm, amg, \ldots\}$.

The *reachability graph* $\mathsf{rg}(EN) = (V, A)$ of EN has V as its set of vertices and A as its set of labelled arcs. V consists of all markings reachable from the initial marking

M_{init}, and A is given as $A = \{(M, U, M') \mid M \in V \wedge M[U\rangle M'\}$. Similarly, the *sequential reachability graph* $\mathsf{rg}_{seq}(EN) = (V', A')$ of EN has V' as its set of vertices and A' as its set of labelled arcs. V' consists of all markings reachable from the initial marking M_{init} through firing sequences, and A' is given as $A' = \{(M, t, M') \mid M \in V' \wedge M[t\rangle M'\}$. It can be seen that although, in general, $\mathsf{rg}_{seq}(EN)$ is a proper subgraph of $\mathsf{rg}(EN)$, their vertices are the same.

The EN-system in Figure 3 is *contact-free* which means that, for all markings M reachable from M_{init} and transitions t, ${}^\bullet t \subseteq M$ implies $t^\bullet \cap M = \varnothing$. Contact-freeness can always be enforced without influencing the step sequence behaviour, by *complementing* (all or some) places p using fresh places $\widetilde{p}$ satisfying ${}^\bullet p = \widetilde{p}^\bullet$, $p^\bullet = {}^\bullet\widetilde{p}$, and declaring that $\widetilde{p} \in M_{init}$ *iff* $p \notin M_{init}$. For example, in Figure 3, $p_4 = \widetilde{p}_3$. In what follows, *all* EN-*systems* as well as their extensions are assumed to be contact-free.

Reachability Graphs and Structure

Strong connections between structure and behaviour have been for a long time a rich source of analytical techniques for Petri nets. These connections are particularly direct in the case of EN-systems. To start with, at a marking M, we say that two transitions, t and v, are:

- *independent*, if they are both enabled and the execution of one does not disable the other. In EN-systems, being independent is equivalent to saying that $\{t, v\}$ is a step enabled at M. This is illustrated in Figure 4(a) for $t = f$ and $v = g$.
- *in conflict*, if they are both enabled and the execution of one disables the other. In EN-systems, being in conflict is equivalent to saying that $\{t, v\}$ is not a step enabled at M. This is illustrated in Figure 4(b) for $t = a$ and $v = f$.
- *causally related*, if one is enabled and its execution makes the other enabled. This is illustrated in Figure 4(c) for $t = m$ and $v = f$.

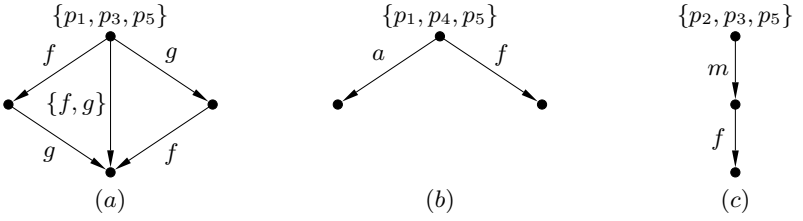


Fig. 4. Independence, conflict and causality in the reachability graph $\mathsf{rg}(EN_0)$

The above relationships are *behavioural*, in the sense that they refer explicitly to executability at markings. There is, however, an alternative characterisation, where we say that two transitions, t and v, are *structurally*:

- *independent*, if ${}^\bullet t^\bullet \cap {}^\bullet v^\bullet = \varnothing$; for example, f and g in EN_0.
- *in conflict*, if ${}^\bullet t \cap {}^\bullet v \neq \varnothing$ or $t^\bullet \cap v^\bullet \neq \varnothing$; for example, f and a in EN_0.
- *causally related*, if $t^\bullet \cap {}^\bullet v \neq \varnothing$ or $v^\bullet \cap {}^\bullet t \neq \varnothing$; for example, m and f in EN_0.

We then obtain a direct connection between the behavioural and structural characterisations of three fundamental relationships between transitions in EN-systems.

Fact 12 (structure vs. behaviour). *In* EN*-systems, behavioural independence, conflict and causality respectively imply structural independence, conflict and causality.*

In other words, transitions which are structurally independent will never be in conflict or causally related whatever the current marking. Similar remarks hold for conflict and causality.

Another observation concerns the relationship between simultaneity and unorderedness in the behaviour of EN-systems. We can formulate the general property that

$$Simultaneity \iff Unorderedness$$

by which we mean that it is always the case that

$$M[\{t,v\}\rangle M' \iff M[tv\rangle M' \wedge M[vt\rangle M' \ .$$

5 Fitting Nets and Order Structures

Given the execution semantics of EN-systems, we could now turn to the development of a causality semantics in terms of occurrence nets and associated causal posets. However, since we aim at a systematic presentation of causality semantics for different net classes, it pays off to develop first a general scheme for doing this. As a result, one can then simplify the formal treatment and also appreciate common properties shared across a range of net classes.

The operational and causality semantics of a class of Petri nets $\mathbb{PN}$ can be presented within a common scheme introduced in [19] (see also [17]) and reproduced here as Figure 5 where N is a net from $\mathbb{PN}$ and:

- $\mathbb{EX}$ are executions (or observations) of nets in $\mathbb{PN}$.
- $\mathbb{LAN}$ are labelled acyclic nets, each representing a concurrent history.
- $\mathbb{LEX}$ are labelled executions of nets in $\mathbb{LAN}$.
- $\mathbb{LCS}$ are labelled causal structures (e.g., order structures) capturing causality relationships between executed actions.

In this paper, $\mathbb{EX}$ will be step sequences, and $\mathbb{LEX}$ labelled singular step sequences. However, $\mathbb{LAN}$ and $\mathbb{LCS}$ will depend on the chosen class of nets $\mathbb{PN}$.

The maps in Figure 5 relate the semantical views captured by $\mathbb{EX}$, $\mathbb{LAN}$, $\mathbb{LEX}$ and $\mathbb{LCS}$:

- ω returns a set of executions, defining the *operational* semantics of N.
- α returns a set of labelled acyclic nets, defining the *axiomatic process* semantics of N.
- π_N returns, for each execution of N, a non-empty set of labelled acyclic nets, defining the *operational process* semantics of N.
- λ returns a set of *labelled* executions for each process of N, and after applying ϕ to such a labelled execution one obtains an execution of N.

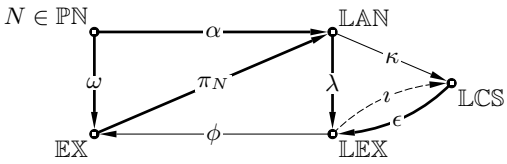


Fig. 5. Semantical framework for a class of Petri nets $\mathbb{PN}$. The bold arcs indicate mappings to powersets and the dashed arc indicates a partial function.

- κ associates a labelled *causal* structure with each process of N.
- ϵ and $\imath$ allow one to go back and forth between labelled causal structures and the sets of their labelled executions.

The semantical framework provided by the schema indicates how the different semantical views should agree. According to the rectangle on the left, the Petri net defines processes satisfying certain axioms and, moreover, all labelled acyclic nets satisfying these axioms can be derived from the executions of the Petri net. Also, the labelled executions of the processes correspond to the executions of the original Petri net. In the triangle on the right, the labelled acyclic nets from $\mathbb{LAN}$, the causal structures from $\mathbb{LCS}$ and the labelled executions from $\mathbb{LEX}$ are related. The order structure defined by a labelled acyclic net can be obtained by combining its executions and, conversely, the stratified extensions of the order structure defined by a labelled acyclic net are the (labelled) executions of that net. Thus the abstract relations between the actions in the labelled causal structures associated with the Petri net will be consistent with its chosen operational semantics.

To demonstrate that these different semantical views agree as captured through this semantical framework, it is sufficient to establish a series of results called *aims*. As there exist four simple requirements (called *properties*) guaranteeing these aims, one can concentrate on defining the semantical domains and maps appearing in Figure 5 and proving these properties.

Property 1 (soundness of mappings). *The maps ω, α, λ, ϕ, $\pi_N|_{\omega(N)}$, κ, ϵ and $\imath|_{\lambda(\mathbb{LAN})}$ are total. Moreover, ω, α, λ, $\pi_N|_{\omega(N)}$ and ϵ always return non-empty sets.*

Property 2 (consistency). *For all $\xi \in \mathbb{EX}$ and $LN \in \mathbb{LAN}$,*

$$\left.\begin{array}{r} \xi \in \omega(N) \\ LN \in \pi_N(\xi) \end{array}\right\} \quad \text{iff} \quad \left\{\begin{array}{l} LN \in \alpha(N) \\ \xi \in \phi(\lambda(LN)) \end{array}\right. .$$

Property 3 (representation). $\imath \circ \epsilon = id_{\mathbb{LCS}}$.

Property 4 (fitting). $\lambda = \epsilon \circ \kappa$.

The above four properties imply that the axiomatic (defined through α) and operational (defined through $\pi_N \circ \omega$) process semantics of nets in $\mathbb{PN}$ are in full agreement. Also, the operational semantics of N (defined through ω) coincides with the operational semantics of the processes of N (defined through $\phi \circ \lambda \circ \alpha$). Finally, the causality in a process of N (defined through κ) coincides with the causality structure implied by its operational semantics (through $\imath \circ \lambda$). That is, we have the following.

Aim 1 $\alpha = \pi_N \circ \omega$.

Aim 2 $\omega = \phi \circ \lambda \circ \alpha$.

Aim 3 $\kappa = \imath \circ \lambda$.

As a consequence, the operational semantics of the Petri net N and the set of labelled causal structures associated with it are related by $\omega = \phi \circ \epsilon \circ \kappa \circ \alpha$.

6 Semantical Framework for EN-Systems

Some of the notions needed to specialise the general concepts of the semantical framework for EN-systems have already been introduced. We will now present the missing ones, starting with the definition of a class of labelled acyclic nets capturing the causality semantics of EN-systems.

An *occurrence net* is a tuple $ON = (P', T', F', \ell)$ such that (P', T', F') is its underlying net[1] and ℓ is a labelling for $P' \cup T'$. Moreover, it is assumed that $|{}^\bullet p| \leq 1$ and $|p^\bullet| \leq 1$, for every place p, and $\varrho_{ON} = (T', (F' \circ F')|_{T' \times T'})$ is a pre-poset (in other words, F' is acyclic). The default *initial* M_{init}^{ON} and *final* M_{fin}^{ON} markings respectively consist of all places without inputs and outputs. Figure 6 shows an occurrence net labelled by places and transitions of the EN-system EN_0 of Figure 3, with the default initial and final markings $\{b_1, b_2, b_3\}$ and $\{b_6, b_9, b_{10}\}$.

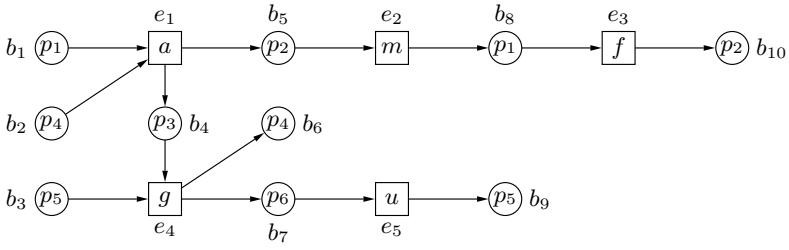


Fig. 6. An occurrence net ON_0 (labels are shown inside the nodes)

Note that, due to the acyclicity of the flow relation and the lack of multiple inputs (or outputs) of places, each transition in T' appears exactly *once* in any step sequence σ satisfying $M_{init}^{ON}[\sigma\rangle M_{fin}^{ON}$. In particular, such a step sequence is singular, and so $\mathsf{spo}(\sigma)$ is a well-defined stratified poset.

The behaviour of an occurrence net ON is captured by the set $\mathsf{steps}(ON)$ of *labelled step sequences*, comprising all pairs $(\sigma, \ell|_{T'})$ such that σ is a step sequence from the default initial marking of ON to the default final marking. For each such labelled step sequence, $\phi(\sigma, \ell|_{T'}) = \ell(\sigma)$. Moreover, $\mathsf{fseq}(ON)$ are the *labelled firing sequences* of ON, i.e., all the labelled step sequences $(\sigma, \ell|_{T'})$ such that σ is a sequence of singleton steps. For the occurrence net of Figure 6, we have $a\{m, g\}\{f, u\} \in \phi(\mathsf{steps}(ON_0))$ as well as $amgfu \in \phi(\mathsf{fseq}(ON_0))$.

[1] The dot-notations, markings, etc, for ON are as those defined for (P', T', F').

Fact 13 (labelled executions). $\mathsf{steps}(ON) \neq \varnothing$ *and* $\mathsf{fseq}(ON) \neq \varnothing$*, for every occurrence net* ON.

For an occurrence net ON, ϱ_{ON} is a pre-poset representing the direct causal relationships between its transitions. Hence, by Fact 4, $\mathsf{po}(ON) = \varrho_{ON}^{\mathsf{po}}$ is the *induced* poset representing all, direct and indirect, causal dependencies between the transitions in T'. For the occurrence net of Figure 6, we have that e_1 causes e_2 directly, but there is only an indirect causal link from e_1 to e_3. Also, there are no causal links between e_3 and e_5 which means that they are independent. This and other relationships can be read out from the diagram of the pre-poset ϱ_{ON_0} shown in Figure 7.

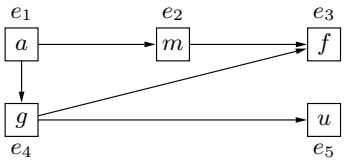


Fig. 7. Pre-poset ϱ_{ON_0} for the occurrence net ON_0

To define processes of an EN-system, we need to provide an axiomatic characterisation of occurrence nets consistent with the structure of this EN-system. A *process* of an EN-system EN is an occurrence net ON with the labelling ℓ which:

- labels places of ON with places of EN.
- labels transitions of ON with transitions of EN.
- is injective on M_{init}^{ON} and $\ell(M_{init}^{ON}) = M_{init}$.
- is injective on ${}^\bullet t$ and $t^\bullet$ and, moreover, $\ell({}^\bullet t) = {}^\bullet \ell(t)$ and $\ell(t^\bullet) = \ell(t)^\bullet$, for every transition t of ON.

We denote this by $ON \in \mathsf{proc}(EN)$. For example, $ON_0 \in \mathsf{proc}(EN_0)$, where EN_0 and ON_0 are the nets in Figures 3 and 6.

Fact 14 (injective labelling). *The labelling* ℓ *of* $ON \in \mathsf{proc}(EN)$ *is injective on any marking reachable from the default initial marking. It is also injective on any individual step appearing in the step sequences of* $\mathsf{steps}(ON) \neq \varnothing$.

The only missing component of the semantical framework for EN-systems is now the mapping returning processes derived from individual step sequences.

The occurrence net $\mathsf{proc}_{EN}(\sigma)$ *generated* by a step sequence $\sigma = U_1 \ldots U_n$ of EN is the last element in the sequence $ON_0, \ldots, ON_n$, where each ON_k is an occurrence net (P_k, T_k, F_k, ℓ_k) constructed in the following way.

Step 0 $P_0 = \{p^1 \mid p \in M_{init}\}$ and $T_0 = F_0 = \varnothing$.
Step *k* Given ON_{k-1}, the nodes and arcs of ON_k are:

$$
\begin{aligned}
P_k &= P_{k-1} \cup \{p^{1+\triangle p} \mid p \in U_k^\bullet\} \\
T_k &= T_{k-1} \cup \{t^{1+\triangle t} \mid t \in U_k\} \\
F_k &= F_{k-1} \cup \{(p^{\triangle p}, t^{1+\triangle t}) \mid t \in U_k \wedge p \in {}^\bullet t\} \\
&\quad \cup \{(t^{1+\triangle t}, p^{1+\triangle p}) \mid t \in U_k \wedge p \in t^\bullet\}\ ,
\end{aligned}
$$

where the label of each node x^i is set to be x, and $\triangle x$ denotes the number of the nodes of ON_{k-1} labelled by x.

The above construction is illustrated in Figure 8 for the EN-system EN_0 of Figure 3. The resulting occurrence net is isomorphic to ON_0 of Figure 6 which, as we already noted, is a process of EN_0.

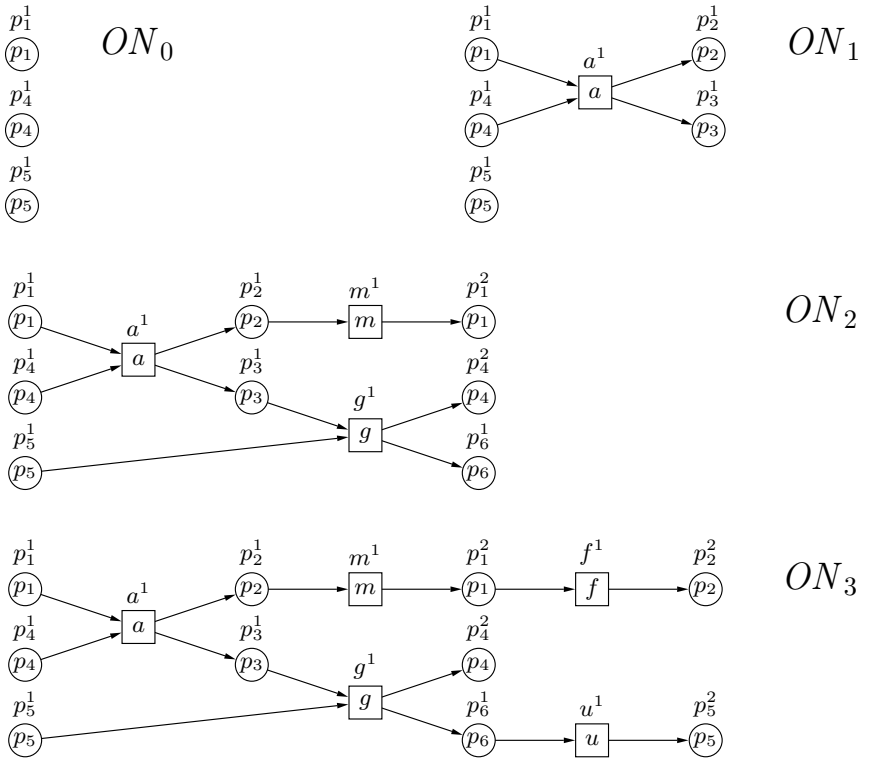


Fig. 8. Process $\mathsf{proc}_{EN_0}(\sigma) = ON_3$ generated for EN_0 and step sequence $\sigma = a\{m, g\}\{f, u\}$

We will now explain how the four semantical properties can be established for EN-systems and their step sequence semantics (the treatment for firing sequences is almost the same). Referring to the notation used in Figure 5, we have the following, where EN is an EN-system, ON an occurrence net, (σ, ℓ) a labelled step sequence, po a poset, and Σ a set of labelled singular step sequences with the same domain:

$\mathbb{PN}$	are	EN-systems	$\mathbb{EX}$	are	step sequences
$\mathbb{LAN}$	are	occurrence nets	$\mathbb{LEX}$	are	labelled singular step sequences
$\mathbb{LCS}$	are	labelled posets			

$\omega(EN)$	is	$\mathsf{steps}(EN)$	$\alpha(EN)$	is	$\mathsf{proc}(EN)$
$\lambda(ON)$	is	$\mathsf{steps}(ON)$	$\pi_{EN}(\sigma)$	is	$\mathsf{proc}_{EN}(\sigma)$
$\phi(\sigma,\ell)$	is	$\ell(\sigma)$	$\kappa(ON)$	is	$\mathsf{po}(ON)$
$\epsilon(po)$	is	$\mathsf{steps}(\mathsf{strat}(po))$	$\imath(\Sigma)$	is	$\bigcap \mathsf{spo}(\Sigma)$.

Properties 1–4 hold for EN-systems [19,27]. Below EN is an EN-system and σ its firing sequence, ON is an occurrence net, po is a poset, and Σ is a set of singular step sequences with the same domain. (Note that Fact 17 follows from Facts 3 and 5.)

Fact 15. $\mathsf{steps}(EN)$, $\mathsf{proc}(EN)$, $\mathsf{steps}(ON)$ *and* $\mathsf{steps}(\mathsf{strat}(po))$ *are non-empty sets. Moreover,* $\mathsf{po}(ON)$ *and* $\bigcap \mathsf{spo}(\Sigma)$ *are posets, and* $\mathsf{proc}_{EN}(\sigma)$ *is an occurrence net.*

Fact 16. $\mathsf{proc}_{EN}(\sigma)$ *is a process of* EN*. Moreover, if* ON *is a process of* EN *and* $\sigma' \in \phi(\mathsf{steps}(ON))$*, then* $\sigma' \in \mathsf{steps}(EN)$ *and* $ON = \mathsf{proc}_{EN}(\sigma')$.

Fact 17. $po = \bigcap \mathsf{spo}(\mathsf{steps}(\mathsf{strat}(po)))$.

Fact 18. $\mathsf{steps}(ON) = \mathsf{steps}(\mathsf{strat}(\mathsf{po}(ON)))$.

Hence we can claim the semantical aims for EN-systems and step sequences.

Fact 19. *Let* EN *be an* EN*-system, and* ON *be an occurrence net.*

$$\begin{aligned} \mathsf{proc}(EN) &= \mathsf{proc}_{EN}(\mathsf{steps}(EN)) \\ \mathsf{steps}(EN) &= \phi(\mathsf{steps}(\mathsf{proc}(EN))) \\ \mathsf{po}(ON) &= \bigcap \mathsf{spo}(\mathsf{steps}(ON))\ . \end{aligned}$$

7 EN-Systems with Activator Arcs

This section extends the treatment of concurrency to nets with activator arcs. Consider again the EN-system of Figure 3 and add an activator arc from place p_4 to transition c with a small black circle as arrowhead. In the resulting net ENA_0 shown in Figure 9, c can only be enabled if there is a token in place p_4. However, the execution of transition c does not consume the token in place p_4.

An *elementary net system with activator arcs* (or ENA-system) is a tuple $ENA = (P, T, F, Act, M_{init})$ such that $\mathsf{und}(ENA) = (P, T, F, M_{init})$ is its underlying EN-system, and $Act \subseteq P \times T$ is a set of *activator* arcs. Notions and notations relating to ENA are inherited from $\mathsf{und}(ENA)$. Moreover, ${}^{\blacklozenge}t$ denotes the set of all the places p where the presence of a token is necessary to enable a transition t, i.e., $(p,t) \in Act$. The behaviour of ENA is also derived from that of $\mathsf{und}(ENA)$ after assuming that a step of transitions U is enabled at a marking M in ENA if it is enabled at M in $\mathsf{und}(ENA)$ and ${}^{\blacklozenge}U \subseteq M$, where ${}^{\blacklozenge}U = \bigcup_{t \in U} {}^{\blacklozenge}t$. The marking resulting from the execution of such a U is exactly the same as it would be in $\mathsf{und}(ENA)$. For the ENA-system of Figure 9, we have that $M[\{a,c\}\rangle M'$ and $M[ca\rangle M'$, where $M = \{p_1, p_4, p_6\}$

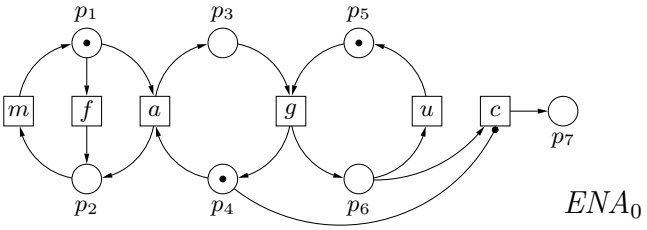


Fig. 9. An ENA-system modelling a second version of the producer/consumer system

and $M' = \{p_2, p_3, p_7\}$. However, $M[ac\rangle M'$ does not hold because after executing transition a, a token is removed from the activator place p_4 of transition c.

Reachability Graphs of ENA-Systems

Reachability in ENA-systems depends on the chosen execution semantics: sequences or step sequences. Taking, as an example the ENA-system in Figure 10(a), we may observe that $M_{init}[\{t, v\}\rangle\{p_3, p_4\}$, but there is no firing sequence σ such that $M_{init}[\sigma\rangle\{p_3, p_4\}$.

Another observation concerns the relationship between simultaneity and unorderedness in the behaviour of ENA-systems. Whereas in the case of EN-systems we have the general property that *Simultaneity* $\Longleftrightarrow$ *Unorderedness* we now have

$$Simultaneity \iff Unorderedness$$

by which we mean that it is always the case that

$$M[\{t, v\}\rangle M' \iff M[tv\rangle M' \wedge M[vt\rangle M' \ .$$

Figure 10(b, c) shows that the reverse implication does not hold.

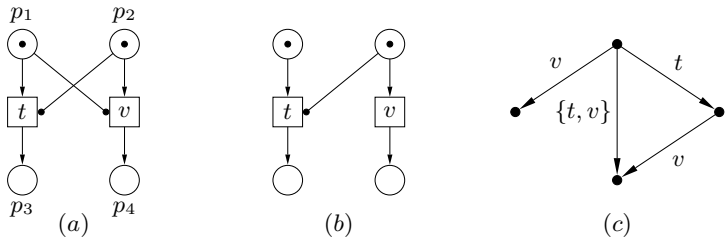


Fig. 10. Two ENA-systems and the reachability graph of the second one

Semantical Framework for ENA-Systems

The causality semantics for ENA-systems will be developed by instantiating the semantical framework, similarly as in the case of EN-systems. The labelled causal structures employed are SO-structures, while executions remain to be (labelled singular) step sequences. To define processes we extend occurrence nets to include activator arcs.

An *activator occurrence net* (or AO-net) $AON = (P', T', F', Act', \ell)$ is a tuple such that $\mathsf{und}(AON) = (P', T', F', \ell)$ is its underlying occurrence net, and $Act' \subseteq P' \times T'$ is a set of *activator* arcs. It is assumed that $\varrho_{AON} = (T', \prec_{loc}, \sqsubset_{loc}, \ell|_{T'})$, where

$$\prec_{loc} = (F' \circ F')|_{T' \times T'} \cup (F' \circ Act') \quad \text{and} \quad \sqsubset_{loc} = (Act')^{-1} \circ F'$$

is a pre-SO-structure (see Figure 11). We then define $\mathsf{sos}(AON) = \varrho^{\mathsf{so}}_{AON}$ to be the SO-structure *induced* by AON.

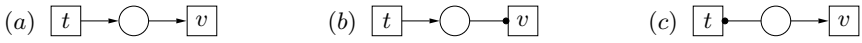


Fig. 11. Two cases (a, b) defining $t \prec_{loc} v$, and one case (c) defining $t \sqsubset_{loc} v$

The step sequences $\mathsf{steps}(AON)$ of an AO-net AON are defined as for $\mathsf{und}(AON)$, except that the enabling condition takes into account activator arcs.

Fact 20 (labelled executions). $\mathsf{steps}(AON) \neq \varnothing$*, for every* AO*-net* ON.

Note that it may happen that $\mathsf{fseq}(AON) = \varnothing$ even though $\mathsf{steps}(AON) \neq \varnothing$. Take, for example, the AO-net AON_1 in Figure 12(a) for which $\mathsf{steps}(AON_1) = \{\{t, v, w\}z\}$ and $\mathsf{fseq}(AON_1) = \varnothing$ as executing at the default initial marking any transition in $\{t, v, w\}$ means that one of the remaining two transitions will never be enabled, and so the default final marking cannot be reached.

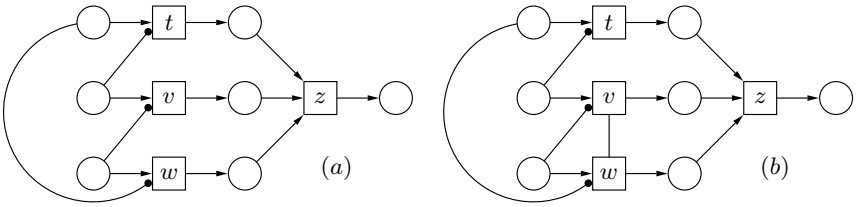


Fig. 12. An AO-net AON_1 (a), and a failed attempt to extend it to an AMO-net (b)

An AO-net represents a concurrent run of a system and has to avoid circularity. Intuitively, $\prec_{loc}$ stands for causal precedence (the first transition has to produce a token for consumption or testing by the second transition) and $\sqsubset_{loc}$ for weak causal precedence (the first transition cannot happen after the second one, since the latter consumes a token which activates the former). Figure 13 shows an AO-net AON_0 labelled by places and transitions of the ENA-system ENA_0 of Figure 9. Its default initial marking is $\{b_1, b_2, b_3\}$, and its default final marking is $\{b_9, b_{10}, b_{11}\}$. Note that transition e_5 weakly precedes transition e_4, i.e., $e_5 \sqsubset_{loc} e_4$. Moreover, we have that $a\{m, g\}\{a, c\}$ and $a\{m, g\}ca$ belong to $\phi(\mathsf{steps}(AON_0))$, but $a\{m, g\}ac$ does not.

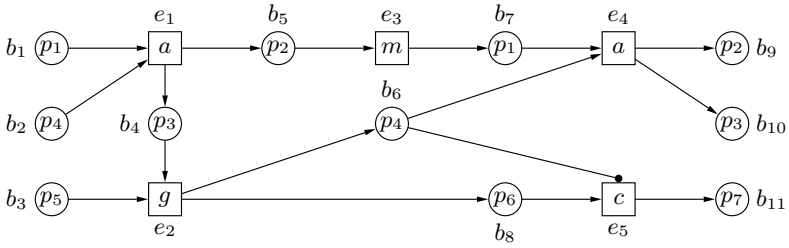


Fig. 13. An activator occurrence net AON_0

Processes of an ENA-system are similar to those of the underlying EN-system extended with an appropriate treatment of activator arcs. A *process* of ENA is an AO-net AON such that $\mathsf{und}(AON)$ is a process of $\mathsf{und}(ENA)$ and, in addition, ℓ is injective on ${}^\blacklozenge t$ and $\ell({}^\blacklozenge t) = {}^\blacklozenge \ell(t)$, for every transition t of AON. We denote this by $AON \in \mathsf{proc}(ENA)$.

Process generation from a given step sequence is also based on that introduced for EN-systems. The AO-net $\mathsf{proc}_{ENA}(\sigma)$ *generated* by a step sequence $\sigma = U_1 \ldots U_n$ of ENA is the last element in the sequence $AON_0, \ldots, AON_n$ where each $AON_k = (P_k, T_k, F_k, Act_k, \ell_k)$ is an AO-net with the components constructed as in the definition for $\mathsf{proc}_{\mathsf{und}(ENA)}(\sigma)$, and the following additions (see Figure 14):

Step 0 $Act_0 = \varnothing$.
Step *k* $Act_k = Act_{k-1} \cup \{(p^{\triangle p}, t^{1+\triangle t}) \mid t \in U \wedge p \in {}^\blacklozenge t\}$.

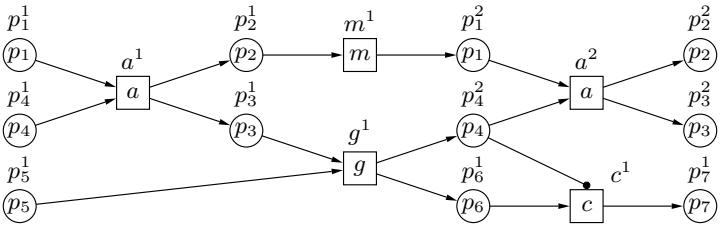


Fig. 14. Process $\mathsf{proc}_{ENA_0}(\sigma)$ generated for ENA_0 and step sequence $\sigma = a\{g, m\}\{a, c\}$

We will now show that the semantical properties formulated in Section 5 can be established for ENA-systems and their step sequences. Referring to the notation used in Figure 5, we have the following, where ENA is an ENA-system, AON an AO-net, (σ, ℓ) a labelled step sequence, sos an SO-structure, and Σ a set of labelled singular step sequences with the same domain:

$\mathbb{PN}$	are	ENA-systems	$\mathbb{EX}$	are	step sequences
$\mathbb{LAN}$	are	AO-nets	$\mathbb{LEX}$	are	labelled singular step sequences
$\mathbb{LCS}$	are	labelled SO-structures			
$\omega(ENA)$	is	$\mathsf{steps}(ENA)$	$\alpha(ENA)$	is	$\mathsf{proc}(ENA)$
$\lambda(AON)$	is	$\mathsf{steps}(AON)$	$\pi_{ENA}(\sigma)$	is	$\mathsf{proc}_{ENA}(\sigma)$
$\phi(\sigma,\ell)$	is	$\ell(\sigma)$	$\kappa(AON)$	is	$\mathsf{sos}(AON)$
$\epsilon(sos)$	is	$\mathsf{steps}(\mathsf{ext}(sos))$	$\imath(\Sigma)$	is	$\bigcap\mathsf{sos}(\mathsf{spo}(\Sigma))$.

It can be shown that Properties 1–4 hold. Below ENA is an ENA-system and σ its step sequence, AON is an AO-net, sos is an SO-structure, and Σ is a set of singular step sequences with the same domain. (Note that Fact 23 follows from Facts 5 and 7.)

Fact 21. $\mathsf{steps}(ENA)$, $\mathsf{proc}(ENA)$, $\mathsf{steps}(AON)$ *and* $\mathsf{steps}(\mathsf{ext}(sos))$ *are non-empty sets. Moreover,* $\mathsf{sos}(AON)$ *and* $\bigcap\mathsf{sos}(\mathsf{spo}(\Sigma))$ *are* SO*-structures, and* $\mathsf{proc}_{ENA}(\sigma)$ *is an* AO*-net.*

Fact 22. $\mathsf{proc}_{ENA}(\sigma)$ *is a process of ENA. Moreover, if* AON *is a process of ENA and* $\sigma' \in \phi(\mathsf{steps}(AON))$*, then* $\sigma' \in \mathsf{steps}(ENA)$ *and* $AON = \mathsf{proc}_{ENA}(\sigma')$.

Fact 23. $sos = \bigcap\mathsf{sos}(\mathsf{steps}(\mathsf{ext}(sos)))$.

Fact 24. $\mathsf{steps}(AON) = \mathsf{steps}(\mathsf{ext}(\mathsf{sos}(AON)))$.

Hence we can claim the semantical aims for ENA-systems.

Fact 25. *Let* ENA *be an* ENA*-system, and* AON *be an* AO*-net.*

$$\begin{aligned}\mathsf{proc}(ENA) &= \mathsf{proc}_{ENA}(\mathsf{steps}(ENA))\\ \mathsf{steps}(ENA) &= \phi(\mathsf{steps}(\mathsf{proc}(ENA)))\\ \mathsf{sos}(AON) &= \bigcap\mathsf{sos}(\mathsf{steps}(AON))\ .\end{aligned}$$

EN-Systems with Inhibitor Arcs

It is easy to extend the treatment presented above for ENA-systems to EN-systems with inhibitor arcs. Consider again the EN-system of Figure 3 and add to it an inhibitor arc linking place p_3 and transition c. This yields the net system ENI_0 shown in Figure 15. (Inhibitor arcs are drawn with small open circles as arrowheads.) Adding such an arc means that c cannot be enabled when the buffer is non-empty (a token in place p_3 signifies that the buffer contains an item).

An *elementary net system with inhibitor arcs* (or ENI-system) is a tuple $ENI = (P, T, F, Inh, M_{init})$ such that $\mathsf{und}(ENI) = (P, T, F, M_{init})$ is its underlying EN-system, and $Inh \subseteq P \times T$ is a set of *inhibitor* arcs. Notions and notations relating to ENI are inherited from $\mathsf{und}(ENI)$. The behaviour of ENI is also derived from that of $\mathsf{und}(ENI)$: a step of transitions U is enabled at a marking M of ENI if it is enabled at M in $\mathsf{und}(ENI)$ and $\{p \mid \exists t \in U : (p,t) \in Inh\} \cap M = \varnothing$. The marking resulting from the execution of such a U is exactly the same as in $\mathsf{und}(ENI)$.

Intuitively, the testing for the presence of tokens using activator arcs in ENA-systems has been replaced by testing for their absence using inhibitor arcs in ENI-systems.

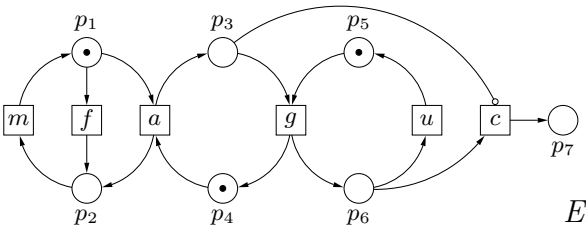


Fig. 15. An ENI-system modelling a third version of the producer/consumer system

In fact, the latter can be faithfully simulated by the former in the case of EN-systems (i.e., they have isomorphic reachability graphs). All we need to assume is that every inhibitor place p has a *complement* place $\widetilde{p}$ satisfying ${}^\bullet p = \widetilde{p}^\bullet$ and ${}^\bullet\widetilde{p} = p^\bullet$. Processes of ENI-systems are similar to those of EN-systems with the inhibitor arcs of the system represented by activator arcs which rather than testing for the absence of tokens are used to test for the presence of tokens in complement places. Hence, we assume that each place p of ENI adjacent to an inhibitor arc has a complement place $\widetilde{p}$ in the underlying EN-system. Then, each inhibitor arc (p, t) can be replaced by an equivalent activator arc $(\widetilde{p}, t)$. Since adding complement places is harmless, we can consider the causality treatment of ENI-systems as being obtained through the corresponding ENA-systems. Note that ENI_0 in Figure 15 corresponds in this way to ENA_0 in Figure 9.

8 ENA-Systems with Mutex Arcs

We now extend ENA-systems with mutex arcs prohibiting certain pairs of transitions from occurring simultaneously (i.e., in the same step). Mutex arcs were introduced in [11], and their causality semantics was developed in [21].

Consider Figure 16 which shows another variant of the producer/consumer scheme. In this case, the consumer is allowed to complete (transition c), but never at the same time as the producer makes an item (transition m). Other than that, there are no restrictions on the executions of transitions c and m. To model such a scenario we use a mutex arc between c and m (depicted as an undirected edge). Note that mutex arcs are relating transitions in a direct way. This should not be regarded as an unusual feature as, for example, Petri nets with priorities also impose direct relations between transitions.

An *elementary net system with activator and mutex arcs* (or ENAM-system) is a tuple $ENAM = (P, T, F, Act, Mtx, M_{init})$ such that $\mathsf{und}(ENAM) = (P, T, F, Act, M_{init})$ is the ENA-system underlying $ENAM$ and $Mtx \subseteq T \times T$ is a symmetric irreflexive relation specifying the *mutex* arcs of $ENAM$. Where possible, we retain the definitions introduced for ENAM-systems. The notion of a step now changes however. A *step of* $ENAM$ is a non-empty set U of transitions such that U is a step of $\mathsf{und}(ENAM)$ and $Mtx \cap (U \times U) = \varnothing$. With this modified notion of a step, the remaining definitions pertaining to the dynamic aspects of an ENAM-system are the same as for the underlying ENA-system.

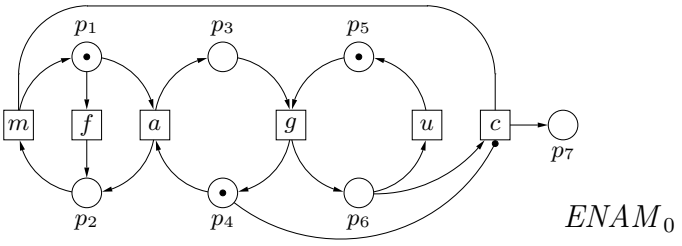


Fig. 16. An ENAM-system modelling a fourth version of the producer/consumer system

For the ENAM-system of Figure 16, we have $M[cm\rangle M'$ as well as $M[mc\rangle M'$, where $M = \{p_2, p_4, p_6\}$ and $M' = \{p_1, p_4, p_7\}$. However, $M[\{c, m\}\rangle M'$ which holds now for the underlying ENA-system does not hold as c and m cannot belong to the same step.

Reachability Graphs of ENAM-Systems

Reachability in ENAM-systems, like in ENA-systems, is affected by the choice of the execution semantics. This is, however, due to the presence of activator arcs, rather than mutex arcs. For an ENAM-system without any activator arcs, the same sets of markings are reachable under the step sequence and firing sequence semantics.

Another observation concerns the relationship between simultaneity and unorderedness in the behaviour of ENAM-systems. Whereas ENA-systems satisfy the relationship *Simultaneity* $\Longleftarrow$ *Unorderedness*, this no longer holds for ENAM-systems, as illustrated in Figure 17.

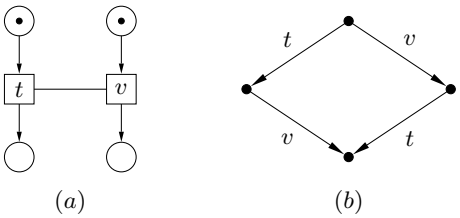


Fig. 17. An ENAM-system without activator arcs and its reachability graph

Semantical Framework for ENAM-Systems

Causality semantics for ENAM-systems will be developed similarly as for EN-systems and ENA-systems. The labelled causal structures employed are GSO-structures, while executions remain to be step sequences. To define processes we extend AO-nets to include mutex arcs. An *activator mutex occurrence net* (or AMO-net) is a tuple $AMON = (P', T', F', Act', Mtx', \ell)$ such that $\mathsf{und}(AMON) = (P', T', F', Act', \ell)$ is the AO-net *underlying* $AMON$ and $Mtx' \subseteq T' \times T'$ is a symmetric irreflexive relation specifying a set of *mutex* arcs. Moreover, it is assumed that

$$\varrho_{AMON} = (T', \prec_{loc}, \sqsubset_{loc}, Mtx') \ ,$$

where $\prec_{loc}$ and $\sqsubset_{loc}$ are defined as for $\mathsf{und}(AMON)$, is a pre-GSO-structure. We then define $\mathsf{gsos}(AMON) = \varrho^{\mathsf{gso}}_{AMON}$ to be the GSO-structure induced by $AMON$. The step sequences $\mathsf{steps}(AMON)$ of $AMON$ are defined as for $\mathsf{und}(AMON)$, except that the definition of a step takes into account mutex arcs. The default initial and final markings of $AMON$, as well as its step sequence executions are defined as for $\mathsf{und}(AMON)$.

The way ϱ_{AMON} deals with the mutex arcs is illustrated in Figure 12(b). We have there three transitions satisfying $t \sqsubset_{loc} v \sqsubset_{loc} w \sqsubset_{loc} t$. Hence, in any execution involving all these transitions, they have to belong to the same step. This, however, is inconsistent with a mutex arc between v and w, and ϱ_{AMON} fails to be a pre-GSO-structure as (t,t) belongs to $\sqsubset^*_{loc} \circ (Mtx' \cap \sqsubset^*_{loc}) \circ \sqsubset^*_{loc}$.

Processes of an ENAM-system are similar to those of the underlying ENA-system extended with appropriate treatment of mutex arcs. A *process* of $ENAM$ is an AMO-net $AMON$ such that $\mathsf{und}(AMON)$ is a process of $\mathsf{und}(ENAM)$ and, in addition, $Mtx' = \{(t,v) \mid (\ell(t),\ell(v)) \in Mtx\}$. We denote this by $AMON \in \mathsf{proc}(ENA)$.

Process generation from a given step sequence is also based on that introduced for EN-systems. The AO-net $\mathsf{proc}_{ENA}(\sigma)$ *generated* by a step sequence $\sigma = U_1 \ldots U_n$ of $ENAM$ is the last element in the sequence $AMON_0, \ldots, AMON_n$, where each $AMON_k = (P_k, T_k, F_k, Act_k, Mtx_k, \ell_k)$ is an AMO-net with the components constructed as in the definition for $\mathsf{proc}_{\mathsf{und}(ENAM)}(\sigma)$ and, in addition:

$$Mtx_k = \{(e,f) \in T_k \times T_k \mid (\ell_k(e),\ell_k(f)) \in Mtx\} \ .$$

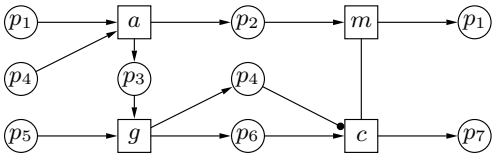


Fig. 18. An AMO-net $AMON_0$

Figure 18 depicts an AMO-net labelled with places and transitions of the ENAM-system of Figure 16. We have that both $agcm$ and $agmc$ belong to $\phi(\mathsf{steps}(AMON_0))$, however, $ag\{m,c\}$ does not. The AMON-net shown in Figure 18 is a process of the ENAM-system of Figure 16 with $\phi(\mathsf{steps}(AMON_0)) = \{agmc, agcm\}$. Figure 19 shows the result of applying the process construction to the ENAM-system of Figure 16 and one of its step sequences. Note that the resulting AMO-net is isomorphic to that shown in Figure 18.

The way in which mutex arcs are added in the process construction means that some may be superfluous. For instance, the transitions they join may be causally related. Analysing paths in the AMO-net would make it possible to eliminate such redundant mutex arcs. This, however, would be against the locality principle which is central to the process approach as it would compromise the local causes and effects in the definition and construction of process nets.

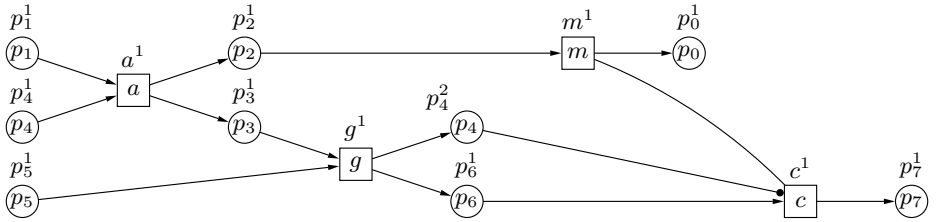


Fig. 19. Process $\mathsf{proc}_{ENAM_0}(\sigma)$ generated for $ENAM_0$ and step sequence $\sigma = \{a\}\{g\}\{m\}\{c\}$

The semantical properties formulated in Section 5 can be established also for ENAM-systems. Referring to the notation used in Figure 5, we have the following, where $ENAM$ is an ENAM-system, $AMON$ an AMO-net, (σ, ℓ) a labelled step sequence, $gsos$ a GSO-structure, and Σ a set of labelled singular step sequences with the same domain:

$\mathbb{PN}$	are	ENAM-systems	$\mathbb{EX}$	are	step sequences
$\mathbb{LAN}$	are	AMO-nets	$\mathbb{LEX}$	are	labelled singular step sequences
$\mathbb{LCS}$	are	labelled GSO-structures			
$\omega(ENAM)$	is	$\mathsf{steps}(ENAM)$	$\alpha(ENAM)$	is	$\mathsf{proc}(ENAM)$
$\lambda(AMON)$	is	$\mathsf{steps}(AMON)$	$\pi_{ENAM}(\sigma)$	is	$\mathsf{proc}_{ENAM}(\sigma)$
$\phi(\sigma, \ell)$	is	$\ell(\sigma)$	$\kappa(AMON)$	is	$\mathsf{gsos}(AMON)$
$\epsilon(gsos)$	is	$\mathsf{steps}(\mathsf{ext}(gsos))$	$\imath(\Sigma)$	is	$\bigcap \mathsf{gsos}(\mathsf{spo}(\Sigma))$.

It can be shown that Properties 1–4 hold. Below $ENAM$ is an ENAM-system and σ its step sequence, $AMON$ is an AMO-net, $gsos$ is an SO-structure, and Σ is a set of singular step sequences with the same domain. (Note that Fact 28 follows from Facts 5 and 10.)

Fact 26. $\mathsf{steps}(ENAM)$, $\mathsf{proc}(ENAM)$, $\mathsf{steps}(AMON)$ *and* $\mathsf{steps}(\mathsf{ext}(gsos))$ *are non-empty sets. Moreover,* $\mathsf{gsos}(AMON)$ *and* $\bigcap \mathsf{gsos}(\mathsf{spo}(\Sigma))$ *are* GSO*-structures, and* $\mathsf{proc}_{ENAM}(\sigma)$ *is an* AMO*-net.*

Fact 27. $\mathsf{proc}_{ENAM}(\sigma)$ *is a process of* $ENAM$*. Moreover, if* $AMON$ *is a process of* $ENAM$ *and* $\sigma' \in \phi(\mathsf{steps}(AMON))$*, then* $\sigma' \in \mathsf{steps}(ENAM)$ *and* $AMON = \mathsf{proc}_{ENAM}(\sigma')$.

Fact 28. $gsos = \bigcap \mathsf{gsos}(\mathsf{steps}(\mathsf{ext}(gsos)))$.

Fact 29. $\mathsf{steps}(AMON) = \mathsf{steps}(\mathsf{ext}(\mathsf{gsos}(AMON)))$.

Hence we can claim the semantical aims for ENAM-systems.

Fact 30. *Let* $ENAM$ *be an* ENAM*-system, and* $AMON$ *be an* AMO*-net.*

$$\begin{aligned} \mathsf{proc}(ENAM) &= \mathsf{proc}_{ENAM}(\mathsf{steps}(ENAM)) \\ \mathsf{steps}(ENAM) &= \phi(\mathsf{steps}(\mathsf{proc}(ENAM))) \\ \mathsf{gsos}(AMON) &= \bigcap \mathsf{gsos}(\mathsf{steps}(AMON)) \ . \end{aligned}$$

9 Place/Transition Nets

Place/Transition nets [26] (or PT-nets) are the basic class of Petri nets suited for the study of systems in which multiplicity of resources matters.

A PT-*net* is a tuple $PT = (P, T, F, M_{init})$ such that (P, T, F) is its *underlying* net, and M_{init} is the *initial* marking of PT, where a marking in this case is any *multiset* of places, i.e., a mapping $M : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. Most notions concerning the structure and graphical representation of PT-nets are the same as for EN-systems except that a marking M is represented by displaying $M(p)$ tokens in each place p. More important changes concern the execution semantics which extends that defined for EN-systems.

A *step* U of PT is any *multiset* of transitions, i.e., $U : T \to \mathbb{N}$. Such a step is *enabled* at a marking M if, for every place p, the current marking M provides enough input tokens for each occurrence of a transition in U, thus $M(p) \geq \sum_{t \in p^\bullet} U(t)$. Executing an enabled step leads to the marking M' such that, for every place p,

$$M'(p) = M(p) - \sum_{t \in p^\bullet} U(t) + \sum_{t \in {}^\bullet p} U(t) \ .$$

We denote this, as before, by $M[U\rangle M'$. The notions of firing sequence, step sequence, marking reachability and reachability graph, are then defined similarly as in the case of EN-systems. Figure 20 depicts three PT-nets such that:

$$\begin{aligned} &\mathsf{fseq}(PT_0) = \{\ldots, amamamam, \ldots\} \\ &M_{init}[gu\{g,a\}\{u,m\}am\rangle M_{init} \qquad (in\ PT_1) \\ &\mathsf{steps}(PT_2) = \{\ldots, a\{g,g\}mama\{u,u\}\{g,g\}, \ldots\} \ . \end{aligned}$$

As in the case of EN-systems, marking reachability in PT-nets does not depend on whether one uses firing sequences or step sequences. This follows from the fact that if U and U' are two steps satisfying $M[U + U'\rangle M'$ then $M[UU'\rangle M'$, where $U + U'$ is the multiset sum of U and U'. As a consequence, every step of transitions occurring at a marking can be split into any sequence of subsets forming a partition of this set, and each such step sequence leads to the same marking as the original step. However, the reverse implication does not, in general, hold. For example, if one takes the PT-net in Figure 23(a), then we have $M_{init}[ab\rangle\{p_2, p_4\}$ and $M_{init}[ba\rangle\{p_2, p_4\}$ but $M_{init}[\{a, b\}\rangle\{p_2, p_4\}$ is not a valid execution. Moreover, the relation between transition occurrences is not structural, but depends on the current marking: with two tokens in p_5 in Figure 23(a), the transitions a and b would be concurrently, i.e., as a step, enabled.

As before, processes formalise the idea of a concurrent run. Interestingly, occurrence nets provide the basis for the process definition of PT-nets in the same way as they did for EN-systems. We only need to take into account the potential multiplicity of tokens in PT-nets. This is done by giving each occurrence of a token its own place in the occurrence nets. A *process* of a PT-net PT is an occurrence net ON with labelling ℓ which:

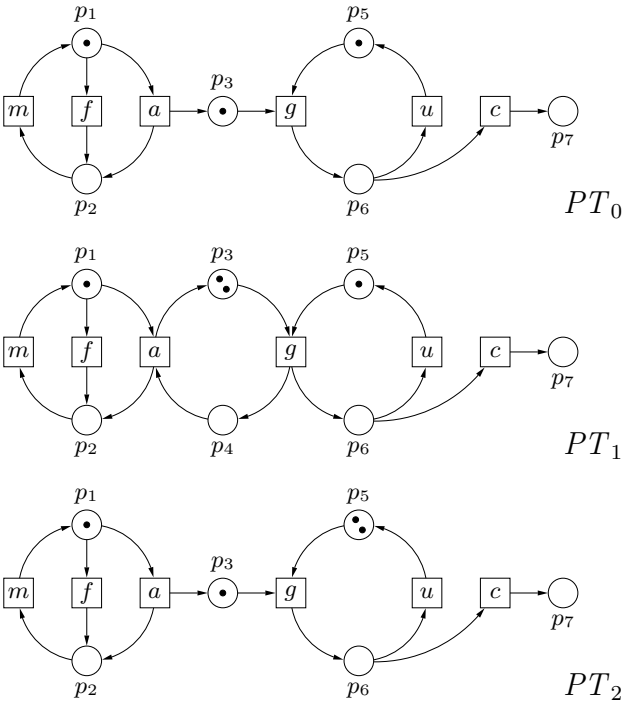


Fig. 20. PT-nets modelling three final versions of the producer/consumer system: PT_0 with an unbounded buffer (the number of tokens in place p_3 can grow unboundedly); PT_1 with a two-place buffer (the number of tokens in place p_3 can be at most two); and PT_2 with an unbounded buffer and two consumers (represented by the two tokens in place p_5)

- labels places of ON with places of PT.
- labels transitions of ON with transitions of PT.
- labels exactly $M_{init}(p)$ places of M_{init}^{ON} with p, for every place p of PT.
- is injective on ${}^\bullet t$ and $t^\bullet$ and, moreover, $\ell({}^\bullet t) = {}^\bullet \ell(t)$ and $\ell(t^\bullet) = \ell(t)^\bullet$, for every transition t of ON.

We denote this by $ON \in \mathsf{proc}(PT)$. The occurrence net ON in Figure 21 is a process of PT-net PT_2 in Figure 20.

The main difference with definition of processes of EN-systems is that now the labelling of a process is not required to be injective on the default initial marking which is meant to represent the initial marking. In general, Fact 14 does not hold for processes of PT-nets. For example, the process in Figure 21 allows the following sequence of executions:

$$\{q_1, q_2, q_3, q_4\}[t_1\rangle\{q_2, q_3, q_4, q_5, q_6\}[\{t_2, t_3\}\rangle\{q_5, q_7, q_8\} \ ,$$

with $\ell(q_7) = \ell(q_8) = p_6$ and $\ell(t_2) = \ell(t_3) = g$.

Defining a process for a given step sequence σ of a PT-net PT is a straightforward extension of the construction for EN-systems. An occurrence net *generated* by a step

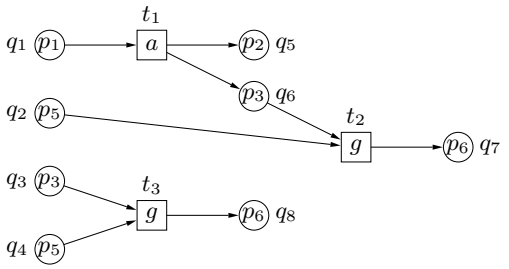


Fig. 21. A process ON of the PT-net PT_2 in Figure 20

sequence $\sigma = U_1 \dots U_n$ of PT is the last element in the sequence $ON_0, \dots, ON_n$, where each ON_k is an occurrence net (P_k, T_k, F_k, ℓ_k) constructed in the following way.

Step 0 $P_0 = \{p^i \mid p \in P \wedge 1 \leq i \leq M_{init}(p)\}$ and $T_0 = F_0 = \varnothing$.
Step *k* Given ON_{k-1}, the nodes of ON_k are given by:

$$P_k = P_{k-1} \cup \{p^{i+\triangle p} \mid p \in P \wedge 1 \leq i \leq \textstyle\sum_{t \in {}^\bullet p} U_k(t)\}$$
$$T_k = T_{k-1} \cup \{t^{i+\triangle t} \mid t \in T \wedge 1 \leq i \leq U_k(t)\} ,$$

where again the label of each node x^i is set to be x, and $\triangle x$ denotes the number of the nodes of ON_{k-1} labelled by x.
To define the arcs, we proceed as follows. For every $e = t^i \in T_k \setminus T_{k-1}$, we *choose* two sets of conditions, $In_e \subseteq M_{fin}^{ON_{k-1}}$ and $Out_e \subseteq P_k \setminus P_{k-1}$, such that In_e comprises a distinct condition p^m for each place $p \in {}^\bullet t$ while Out_e comprises a distinct condition q^l for each place $q \in t^\bullet$. Moreover, for any $e \neq f \in T_k \setminus T_{k-1}$, $In_e \cap In_f = \varnothing$ and $Out_e \cap Out_f = \varnothing$. Then:

$$F_k = F_{k-1} \cup \bigcup_{e \in T_k \setminus T_{k-1}} (In_e \times \{e\}) \cup (\{e\} \times Out_e) .$$

We denote this by $ON_n \in \mathsf{proc}_{PT}(\sigma)$.

Note that since there may be more than one choice of suitable In_e's, in general, more than one process can be constructed for a given step sequence σ. The above construction is illustrated in Figure 22 for PT-net PT_2 of Figure 20. The resulting occurrence net is isomorphic to ON of Figure 21 which, as we already noted, is a process of PT_2.

The detailed development of the process semantics of PT-nets can be carried out along the same lines as was done for EN-systems earlier in this paper, with some straightforward modification resulting from the multiset — rather than set — nature of markings and executed steps. It is also possible to extend the treatment of PT-nets to include weighted arcs and (weighted) activator and inhibitor nets, using AO-nets as a process model, following what was done for ENA-systems and ENI-systems (see, e.g., [19,20]).

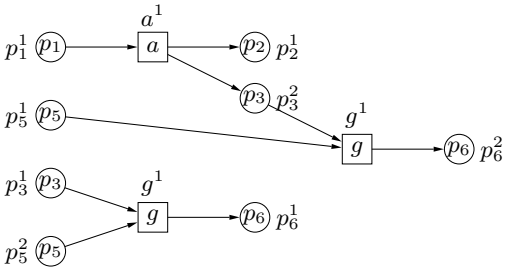


Fig. 22. Deriving a process for PT_2 and its step sequence $\sigma = \{a, g\}g$

Mutex Arcs and Self-Loops

In PT-nets, in contrast to EN-systems, mutex arcs can be represented by self-loops connected to a place marked with a single token, as shown in Figure 23(a, b). From a modelling perspective, there appears to be no real difference. Semantically, however, the differences can be significant as mutex arcs represent concurrent histories in a more compact way. This could have an impact on net unfoldings used for model checking. For example, the single process in Figure 23(c) derived for the representation of Figure 23(b) has to be replaced by two processes derived for the representation of Figure 23(a) depicted in Figure 23(d).

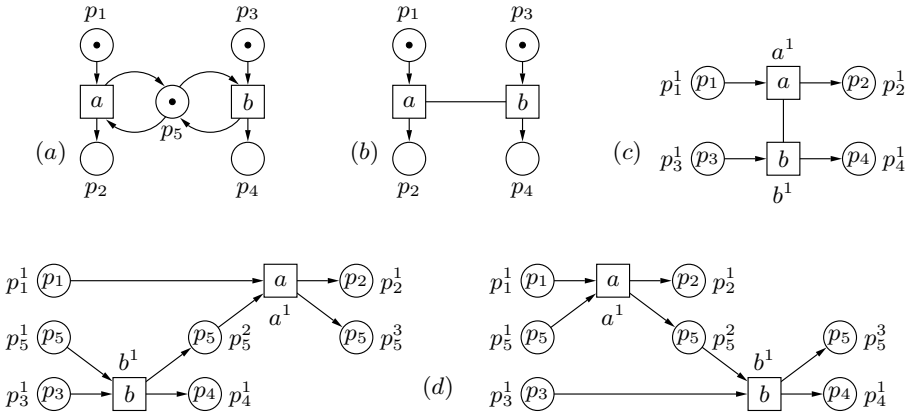


Fig. 23. Mutex arcs can lead to more condensed process semantics than self-loops

10 Concluding Remarks

This paper is an introduction to the many issues fundamental to understanding concurrent behaviour. Here we have concerned ourselves with different forms of causality induced by extensions to the basic structure of Petri nets and leading to relational structures extending the classical partial order approach. There are several strands of related

research which have not been described here. For instance, we have not considered the modelling of conflicts between enabled transitions. Our processes and their abstractions (partial orders) model concurrent runs in which conflicts have already been resolved. Branching processes of Petri nets [3] model all possible choices and lead to a single unfolding representing all runs of the net model. They are actually the basis for efficient verification techniques [5,18,23]. If, in addition, one abstracts from state information and only considers relations between events, the result is the more abstract model of event structures [9,25,29], that can be used to study fundamental concepts of concurrency in a model-independent way. As far as we are aware, event structures have not yet been enriched with weak causality and commutativity relationships, and we consider such extensions a relevant, and indeed exciting, topic of future research in this area.

Finally, an abstraction not considered here at all, usually referred to as trace theory [2] initiated in [22], allows one to group together sequential observations on the basis of reordering of concurrent (independent) events. The resulting model of *trace monoid* captures precisely the semantical treatment of EN-systems outlined in this paper. For the extended models of ENI/ENA-systems, one needs to use the extended model of *comtraces* introduced in [14]. The last extension of EN-systems considered here, i.e., ENAM-systems, calls for the even more elaborate model of *generalised comtraces* [16]. It should then not come as a surprise that PT-nets require a different kind of extensions of the basic trace monoid, initiated through the work on local traces of [8]. An extensive account of the intrinsic relationships between various concurrency monoids and different net classes can be found in [11].

References

1. Best, E., Devillers, R.: Sequential and concurrent behaviour in Petri net theory. Theor. Comput. Sci. 55(1), 87–136 (1987)
2. Diekert, V., Rozenberg, G. (eds.): The Book of Traces. World Scientific, Singapore (1995)
3. Engelfriet, J.: Branching processes of Petri nets. Acta Inf. 28(6), 575–591 (1991)
4. Esparza, J., Heljanko, K.: Unfoldings: A Partial Order Approach to Model Checking. Monographs in Theoretical Computer Science. Springer (2008)
5. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 87–106. Springer, Heidelberg (1996)
6. Gaifman, H., Pratt, V.R.: Partial order models of concurrency and the computation of functions. In: LICS, pp. 72–85. IEEE Computer Society (1987)
7. Guo, G., Janicki, R.: Modelling concurrent behaviours by commutativity and weak causality relations. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 178–191. Springer, Heidelberg (2002)
8. Hoogers, P.W., Kleijn, H.C.M., Thiagarajan, P.S.: A trace semantics for Petri nets. Inf. Comput. 117(1), 98–114 (1995)
9. Hoogers, P.W., Kleijn, H.C.M., Thiagarajan, P.S.: An event structure semantics for general Petri nets. Theor. Comput. Sci. 153(1&2), 129–170 (1996)
10. Janicki, R.: Relational structures model of concurrency. Acta Inf. 45(4), 279–320 (2008)
11. Janicki, R., Kleijn, J., Koutny, M.: Quotient monoids and concurrent behaviour. In: Martín-Vide, C. (ed.) Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory, vol. 2, pp. 313–386. World Scientific (2010)

12. Janicki, R., Koutny, M.: Invariants and paradigms of concurrency theory. In: Aarts, E.H.L., van Leeuwen, J., Rem, M. (eds.) PARLE 1991. LNCS, vol. 506, pp. 59–74. Springer, Heidelberg (1991)
13. Janicki, R., Koutny, M.: Structure of concurrency. Theor. Comput. Sci. 112(1), 5–52 (1993)
14. Janicki, R., Koutny, M.: Semantics of inhibitor nets. Inf. Comput. 123(1), 1–16 (1995)
15. Janicki, R., Koutny, M.: Fundamentals of modelling concurrency using discrete relational structures. Acta Inf. 34(5), 367–388 (1997)
16. Janicki, R., Le, D.T.M.: Modelling concurrency with comtraces and generalized comtraces. Inf. Comput. 209(11), 1355–1389 (2011)
17. Juhás, G., Lorenz, R., Mauser, S.: Complete process semantics of Petri nets. Fundam. Inform. 87(3-4), 331–365 (2008)
18. Khomenko, V., Koutny, M., Vogler, W.: Canonical prefixes of Petri net unfoldings. Acta Inf. 40(2), 95–118 (2003)
19. Kleijn, H.C.M., Koutny, M.: Process semantics of general inhibitor nets. Inf. Comput. 190(1), 18–69 (2004)
20. Kleijn, J., Koutny, M.: Processes of Petri nets with range testing. Fundam. Inform. 80(1-3), 199–219 (2007)
21. Kleijn, J., Koutny, M.: The mutex paradigm of concurrency. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 228–247. Springer, Heidelberg (2011)
22. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. Tech. Rep. DAIMI PB 78, Aarhus University (1977)
23. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993)
24. Montanari, U., Rossi, F.: Contextual nets. Acta Inf. 32(6), 545–596 (1995)
25. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theor. Comput. Sci. 13, 85–108 (1981)
26. Reisig, W., Rozenberg, G. (eds.): Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996, Lecture Notes in Computer Science, vol. 1491. Springer (1998)
27. Rozenberg, G., Engelfriet, J.: Elementary net systems. In: Reisig, Rozenberg (eds.) [26], pp. 12–121
28. Szpilrajn, E.: Sur l'extension de l'ordre partiel. Fundam. Mathem. 16, 386–389 (1930)
29. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)

External Behaviour of Systems of State Machines with Variables

Antti Valmari

Tampere University of Technology, Department of Mathematics
P.O. Box 553, FI-33101 Tampere, Finland
Antti.Valmari@tut.fi

Abstract. This tutorial is an introduction to compositionality and externally observable behaviour. To make it easier to understand system descriptions, traditional process-algebraic languages have been replaced by state machines represented as annotated directed graphs. Emphasis is on a novel way of treating local variables, and on the Chaos-Free Failures Divergences semantics. Even so, big themes that are not tied to any particular semantics are pointed out where possible. Other semantic models are introduced briefly. Most important verification methods facilitated by compositionality are mentioned with pointers to literature. Mathematical details are given less attention but not left out altogether. Throughout the tutorial, important principles are summarized in framed pieces of text.

1 Introduction

External behaviour, or *externally observable behaviour*, is the behaviour of an entity as seen at its interface, without seeing inside the entity or its other interfaces. For instance, when withdrawing cash from an automated teller machine (ATM), the user enters her bank card into the slot, types something, gets or does not get money, and gets the card back, and later she sees from the statement that the balance of her account has reduced accordingly. The user does not see the telecommunication between the ATM and the central computer of the bank, and she does not see that the computer checked whether her account had enough money. For another instance, a C++ program sees the C++ standard library `std::map` as something to which (key, value)-pairs can be added, accessed via the key, and removed, but the program does not see the red/black-tree operations that take place inside.

Computer science and software engineering favour abstraction and the description of things in implementation-independent ways. In the case of data structures, this desire has led to the development of abstract data types and specification methods for them, such as algebraic data types. Things become much more difficult with concurrent systems (we will see in Sect. 5.7 that nondeterminism is the culprit). This has led to the development of hundreds of different notions of external behaviour of concurrent systems. Fortunately, there are many common ideas and unifying themes. This tutorial is an introduction

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 255–299, 2013.

to central insights in theories of the externally observable behaviour of concurrent systems. Many of the presented ideas are very well-known, but we will also discuss some that seem less well-known.

Every theory and every tutorial has its limits. This one concentrates on interleaving concurrency and lacks the aspects of real-time and probabilities. One might also complain that only the process-algebraic view is presented. However, this is because there seem to be no alternatives: the vast majority of articles on the external behaviour of concurrent systems are either openly or implicitly based on process-algebraic ideas. It seems to the present author that although the process-algebraic languages are definitely artificial and can be replaced by other notations (as we will do), at the semantic level process algebras have found something universally valid. On the other hand, although the theory of recursive process expressions has received a lot of attention in the literature, it is essentially absent from this tutorial. This is because it is not needed for discussing compositionality and external behaviour. It is also quite difficult.

There are many extensive treatments on process algebras, including [13,20,23,25]. Also the present author has published two tutorials [30,34] and touched the topic in [32]. It is reasonable to ask whether the world needs yet another one. After reading many papers and submissions over the years, it seems to the present author that many researchers try to re-invent results in the field, without realizing that a lot exists already. Perhaps this is because external behaviour, and its close friend compositional analysis, are very natural and desirable goals; but much of the literature on process algebras seems, at the first sight, to present hard theories with a narrow scope instead of material that the reader could apply to her own situation.

This suggests that there is a need for a tutorial that presents the big picture or roadmap in an easily accessible way, without *unnecessarily* requiring its readers to dwell in tricky details that process-algebraic theories abound. (This does not mean that tricky details could be avoided altogether.) This tutorial tries to be such. The readers will decide whether it succeeds.

Writing this tutorial also gave the chance to discuss some ideas that have received little attention although the present author believes that they are fundamental. Finally, the treatment of local variables of state machines in this tutorial has not been published before, excluding some sketchy preliminary versions. It is largely similar to well-known approaches, but uses so-called "data manipulation relations" to separate semantics from syntax. It is therefore given a lot of attention, while material that can be found in earlier tutorials is discussed more briefly.

Big themes in a research field tend to become apparent gradually. Often there is no well-defined first paper, where some idea has first been presented in a clear form. For instance, after many enough papers it has just become more or less common knowledge that alphabet-based synchronization is "the" fundamental notion of parallel composition, because it is simple and (as we will see in Sect. 4) universal in the sense that other common parallel composition operators can be constructed from it, but not always vice versa. For this reason, this tutorial does

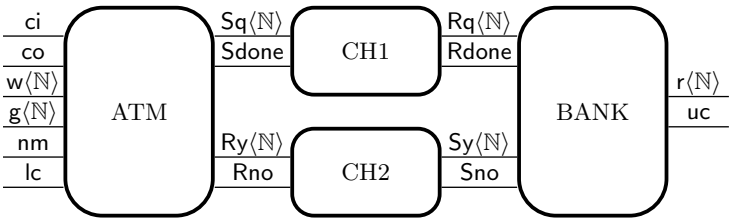


Fig. 1. A simplified cash dispenser system

not contain many references. Some of the given references point to recommended further reading and not necessarily to the original publications.

Section 2 presents the formalism that we will use for describing concurrent systems: state machines with local variables. It also presents a running example that will be used throughout this tutorial. The behaviour of a state machine is the topic of Sect. 3. The section also covers what it means for two behaviours to be equivalent at a detailed level. In Sect. 4 we will discuss the composition of a system from interacting state machines and the behaviour of the result. Behaviour at an abstract level is a central topic in process algebras. It is discussed in Sect. 5. The most well-known semantic models are introduced and CFFD-semantics is discussed in more detail. The section also briefly introduces verification techniques related to compositionality. Section 6 comments on the state of the art. Throughout the tutorial, important principles are informally summarized in boxes.

2 State Machines

In this tutorial, systems are presented as collections of interacting state machines. In this section we concentrate on individual state machines. We first illustrate them with the aid of an example system and then present a formal definition. In the meantime, we also comment on subtleties in the operation of the example system. Finally, we briefly introduce an issue that is of secondary importance for this tutorial but very important in the verification of concurrent systems in general: state propositions.

2.1 A Cash Dispenser System

Our example system is a simple cash dispenser system. Let us first discuss its overall design and operation. The system is shown in Fig. 1. It consists of an automated teller machine (ATM), a bank computer, and telecommunication channels between them. It models how a user can withdraw money and how that affects the balance of her account. To keep the example simple, we only model one user and account, and leave out many details such as user authentication.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote the set of natural numbers. The operation starts when the user puts in her bank card (ci for card in). Then she types the amount

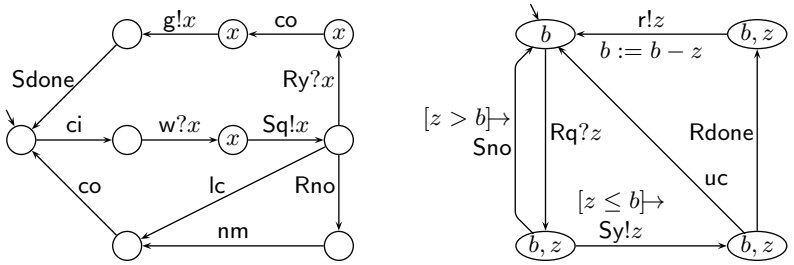


Fig. 2. The ATM and BANK state machines

of money she wants to withdraw ($\mathsf{w}\langle\mathbb{N}\rangle$). The amount is represented as a natural number parameter of w. Such parameters are called *event parameters* in this tutorial. Event parameters are thus the values that are communicated between state machines and/or the environment of the system during an event.

ATM sends the bank a query whether the user can withdraw that much money ($\mathsf{Sq}\langle\mathbb{N}\rangle$). CH1 either delivers the query to the bank ($\mathsf{Rq}\langle\mathbb{N}\rangle$) or loses it. The bank computer checks the balance of the account and sends back the answer "yes" or "no" via CH2. The yes-answer carries a number indicating the amount of money for a reason that we will explain in Sect. 2.3. Also CH2 may lose messages.

Depending on the answer, ATM either gives the money to the user ($\mathsf{g}\langle\mathbb{N}\rangle$) or replies that the user does not have enough money (nm). To prepare for losses of messages, ATM has a timeout mechanism that may make it abort the transaction and tell the user that connection was lost (lc). In any case, ATM gives the user the card back (co). If ATM gave the user the money, it informs the bank by sending the message "done". When the bank receives it, it reduces the balance of the account accordingly ($\mathsf{r}\langle\mathbb{N}\rangle$). If the bank waits for "done" in vain, a timer triggers and makes the bank record that the outcome of the transaction is uncertain (uc) and someone must go to the real physical ATM to check its transcript. The rationale of these details will be discussed in Sect. 2.3.

2.2 State Machines of the Cash Dispenser System

Figure 2 shows the ATM and BANK state machines. Our notion of a state machine is pretty much like a coloured Petri net with precisely one token and, consequently, precisely one input and output arc for each Petri net transition. The states (drawn as circles or ovals) correspond to Petri net places and the transitions (arrows) correspond to Petri net transitions and arcs. Together they show when the ci, $\mathsf{w}\langle 20\rangle$, etc., events can take place. The initial state is indicated with a short arrow that does not start at a state. It corresponds to the Petri net place where the only token is initially.

Figure 1 has $\mathsf{w}\langle\mathbb{N}\rangle$, because it does not tell who decides the amount of money that is withdrawn, while Fig. 2 has $\mathsf{w}?x$ to indicate that ATM is ready for just any amount, and that amount will be stored in variable x of the next state.

Fig. 3. The CH1 and CH2 state machines

The execution of a transition is called *event*. In an event the former syntax is used and $\mathbb{N}$ has been replaced by a natural number, like in $\mathsf{w}\langle 20\rangle$. In the case of $\mathsf{g}!x$ the amount of money is determined by ATM and is, of course, the same as the value of x in the preceding state. In terms of coloured Petri nets, roughly speaking, in the case of ! the variable is present in the inscription of the input arc of the Petri net transition, while ? corresponds to the output arc. We will discuss ! and ? in more detail in Sect. 4.1.

The variable x only exists in some states. This is not fundamental, because we could extend the type of x by an additional value "undefined" and let x have that value in the remaining states, or we could just let x keep its most recent value when its value does not matter. On the other hand, drawing x into only some states makes it explicit when the value of x is and is not significant. This helps sometimes understanding systems. The issue is similar to a coloured Petri net token having some data component in some places and not having it in some other places. We will explain in Sect. 2.3 why x is not present in the start state of the lc-transition.

BANK has two variables, b for the balance and z for temporary storage. The notation $[z > b]\!\!\rightarrow$ indicates that the transition is possible only if $z > b$, that is, the user asked for more money than her bank account has. Such conditions are called *guards*, and they correspond to guards in coloured Petri nets. Obviously $b := b - z$ models the updating of the balance. In coloured Petri nets, the same effect is obtained by specifying the value of a component of an outgoing token with a function on the output arc.

Figure 3 shows the channels. Here our notation is not optimal, because we had to draw the same theme twice in CH1, once for $\mathsf{q}\langle\mathbb{N}\rangle$ and once for done. However, this is notational inconvenience and not important for our goal that emphasizes semantics. The label τ is a special label that denotes that the execution of the transition is not directly observable by anything outside CH1. We say that τ is the *invisible action*. Here τ-transitions model losses of messages. CH2 is similar. It could be made from CH1 with the renaming operator of Sect. 4.3.

The examples demonstrate that state machines may have local variables and sequential computation with them. There are no shared variables in our formalism. This does not imply loss of generality, because one can represent shared variables as state machines with local variables. Indeed, CH1 and CH2 are examples of this. We will see an even more direct example in Fig. 7.

Interaction and communication between state machines belongs to Sect. 4.1, but the basic principle must be told here to be able to discuss the behaviour of the cash dispenser system as a whole. We say that ci, w, and so on are *gates*. Every state machine has an associated set of gates, and many state machines

may share the same gate. In Fig. 1, the gates of the cash dispenser system are shown by lines and their labels (excluding the "⟨ℕ⟩"-parts).

To execute a transition whose label is or starts with a gate name, all state machines that are connected to that gate must execute a transition with that gate name simultaneously, and the numbers and values of event parameters must match. For instance, if ATM wants to execute $\mathsf{Sq}\langle 10\rangle$, then also CH1 must execute $\mathsf{Sq}\langle 10\rangle$. If CH1 is not ready to execute it, then ATM cannot execute it either. On the other hand, if a transition is labelled with τ, then it is executed by that state machine alone.

In terms of coloured Petri nets, this resembles the fusion of transitions that have the same gate name, except that τ-transitions are not fused. Furthermore, fusion is made in all combinations where precisely one transition with the given gate name is picked from each participating state machine. So, if A has two a-transitions, B has four, and C has three, then there are $2 \cdot 4 \cdot 3 = 24$ fused a-transitions. Finally, event parameters do not have a direct counterpart, although similar effects may be obtained by using the same coloured Petri net variable name in more than one fused transition.

The set of gates of a state machine must contain all gates referred to by the transitions of the state machine, and it may contain more. The presence of unused gates in the set of gates is significant, because it prevents the neighbour state machines from executing transitions with those gate names. One may argue whether this is an undesirable feature of the theory, but at least it is so difficult to change that it is better to leave it like that. This is because if the set of gates were defined as the set of used gates, one could cheat by adding an extra, always disabled transition with any desired gate name. It is more transparent to allow the user put extra gates to the set of gates if she wishes.

2.3 Remarks about the Behaviour of the Cash Dispenser System

We already discussed the two main sequences of events of the cash dispenser system: successful withdrawal of money and failure because of not having enough money. We also pointed out that messages may be lost, but ATM recovers from that by executing lc and BANK recovers with uc. The latter deserves a comment.

Ideally, we would like BANK to always get the right idea about whether the money was given to the user. Unfortunately, sending "yes" to ATM does not guarantee that the money is given, because "yes" may be lost in the channel, causing ATM to report loss of connection and not give the money. The "done" message prevents BANK from incorrectly reasoning that the money is given, but introduces the possibility of the opposite type of error, that is, incorrectly reasoning that the money was not given. It is common knowledge in telecommunication that in this kind of a situation, wrong conclusions cannot be fully avoided if a protocol that always eventually terminates is used. Wrong conclusions can be fully avoided if BANK may ask ATM about the outcome again and again until it receives an answer, but that may lead to a never-ending sequence of messages.

Fortunately, we can rule out one of the two types of wrong conclusions. We chose to prevent the one where ATM does not give the money but the account is charged. Furthermore, when the system gives the money without charging the account, the warning uc is always issued, so the people in the bank can go to the physical ATM, check the situation, and fix the balance afterwards. Unfortunately, there will also be false uc alarms.

Another subtle issue is the presence of the amount in the "yes" message and its absence in the start state of the lc-transition of ATM. Their purpose is to prevent the following and similar sequences of events. The user starts a transaction. The BANK computer is busy, and sends "yes" so slowly that the timer in ATM triggers and ATM executes lc before the "yes" arrives. The user reads the reply by ATM carelessly and thinks that she tried to withdraw too much money. Therefore, after getting the card back, she puts the card in again and types a new, smaller amount to withdraw. ATM sends the corresponding query, but it is lost. Next the delayed "yes" message arrives. If it lacked the amount information, ATM would interpret it as a "yes" to the new amount. So it gives the card and money, and sends "done". BANK receives "done" and charges the original amount from the account, which is different from what was given to the user.

So the purpose of the presence of the amount in the "yes" messages is to guarantee that the given amount is always the same as the charged amount (if it is charged). The start state of the lc-transition does not need the amount, because it is picked from the "yes" message.

The subtleties discussed above may make the reader wonder whether we have ruled out all errors. We have not, but fortunately there are verification tools.

2.4 Formal Definition of State Machines

In this subsection we will define state machines formally. To avoid having to define the language used in the guards and assignments of transitions, and to get a much more general theory than allowed by the simple notation that we have discussed, we will introduce abstract *data manipulation relations*. The !-, ?-, $[\cdots]\mapsto$-, and :=-notation will be interpreted as a handy practical representation for a subset of the data manipulation relations, useful for presenting examples.

Each state machine has the set of *types* used by it, denoted with Θ. Formally, a type is just a nonempty set. For instance, the set of natural numbers is a commonly used type. Each variable of each state has a type from Θ.

It would seem natural to give a type also to each event parameter of each transition, and we often do so in examples. However, it will become evident in Sect. 4 that it is better that the formal definition does not pay attention to the internal structure of the labels of events. Therefore, labels of events will be formalized as arbitrary symbols called *actions*, and types will not be needed for that. We have already mentioned that the invisible action τ has a special role. The remaining actions are *visible*. The set of them will be called the *alphabet* and denoted with Σ. We stipulate that $\tau \notin \Sigma$.

For instance, the alphabet of BANK could usually in practice be chosen as $\{\, \mathsf{r}\langle 0\rangle, \mathsf{r}\langle 1\rangle, \mathsf{r}\langle 2\rangle, \ldots, \mathsf{uc}, \mathsf{Rq}\langle 0\rangle, \mathsf{Rq}\langle 1\rangle, \ldots, \mathsf{Rdone}, \mathsf{Sy}\langle 0\rangle, \mathsf{Sy}\langle 1\rangle, \ldots, \mathsf{Sno} \,\}$. However, this relies on the assumption that all transitions of all state machines have the correct number and types of event parameters. To see what might go wrong if this does not hold, assume that there is also a state machine that represents the user of the cash dispenser system. The user can execute $\mathsf{g}\langle 20\rangle$, that is, get 20 units of money, only when ATM is ready for that. However, if $\mathsf{g}\langle 20.5\rangle$ is in the alphabet of the user but not in the alphabets of the other state machines, then the user can get 20.5 units of money any time at will! Therefore, in the presence of event parameters, the precise alphabet of a state machine is $\Gamma \times \mathcal{U}^*$, where Γ denotes the set of gates and $\mathcal{U}$ denotes the set of all data values used by the system.

Referring to the names of variables in the formal definition would be clumsy. Therefore, for each state s, we assume that its variables are listed in some order and rely on their positions in this order. The types of the variables of s can now be specified as a Cartesian product $\mathcal{T}(s)$ of types. For instance, if s has variables n, b, and x of types `int`, `bool`, and `float`, and if we choose to list them in this order, then $\mathcal{T}(s) = \texttt{int} \times \texttt{bool} \times \texttt{float}$. To handle states that have no variables, we denote the empty list of variable values with $\langle\rangle$. Thus, if s has no variables, then $\mathcal{T}(s) = \{\langle\rangle\}$. With $\Theta^\times$ we denote the set of all finite Cartesian products of types, that is, the set of all $T_1 \times \cdots \times T_n$ where n is a natural number and $T_i \in \Theta$ for $1 \le i \le n$. So $\mathcal{T}(s) \in \Theta^\times$, and $\{\langle\rangle\}$ is the element of $\Theta^\times$ that results from $n = 0$.

Of course, the state machine has a *set of states* S and an *initial state* $\hat{s}$. The *initial values* of the variables of the initial state are listed by $\hat{v}$. So $\hat{v} \in \mathcal{T}(\hat{s})$. In some applications, more than one initial state or more than one possible initial value for a variable would be useful, but we do not present that possibility in our formal definition, to avoid making it more complex.

The most complicated part is the set Δ of *transitions*. Each transition is a tuple (s, R, s'), where s is the start state and s' is the end state of the transition. R is the data manipulation relation and will be discussed next.

Let $v_1, \ldots, v_n$ denote the values of the variables of s before executing the transition. Similarly, let $w_1, \ldots, w_m$ be the values of the variables of s' after the transition. So $\langle v_1, \ldots, v_n\rangle \in \mathcal{T}(s)$ and $\langle w_1, \ldots, w_m\rangle \in \mathcal{T}(s')$. Let a be the action, that is, the label of the event. When the ?-, etc., notation is used, a is the gate name together with the values of the event parameters. The purpose of R is to express the dependency between v_1, ..., v_n, a, and w_1, ..., w_m. For instance, in the case of $\mathsf{a}!(v_2 + 1)?w_5$, R must say that $a = \mathsf{a}\langle p_1, p_2\rangle$, where $p_1 = v_2 + 1$ and $w_5 := p_2$. If the transition also has the guard $[v_1 \ne 1]\!\rightarrow$ and the assignment $w_4 := 2v_1$, then R must also reflect their effect.

A natural first idea would be to let R be a collection of partial functions of the gate name, $v_1, \ldots, v_n$, and those event parameters that are specified with ?. The functions would yield the values of all w_i and the remaining event parameters. They would be partial because of guards. However, sometimes modellers need nondeterministic operations, like the assignment of a random value to a variable.

Furthermore, some process-algebraic languages also feature conditions that test the inputted values. Adding these facilities to partial functions would make the formalism complicated. It is easier — and also more general — to let R be an arbitrary relation.

Therefore, R is defined as a relation on $\mathcal{T}(s) \times (\Sigma \cup \{\tau\}) \times \mathcal{T}(s')$. Assume that the state machine is in state s, and the values of the variables of s are $v_1, \ldots, v_n$. The state machine is ready to execute the transition with event name a to state s' yielding its variables the values $w_1, \ldots, w_m$ if and only if $R(v_1, \ldots, v_n, a, w_1, \ldots, w_m)$ holds. To avoid clumsy formulas, we abbreviate $v_1, \ldots, v_n$ and $w_1, \ldots, w_m$ to $\bar{v}$ and $\bar{w}$. There may be many a and $\bar{w}$ with which $R(\bar{v}, a, \bar{w})$ holds with the current values of the v_i. Most importantly, $?w_5$ at event parameter position 2 allows many values for the second event parameter and w_5, as long as they both have the same value. The transition may thus have many instances. Many instances of the same transition is the same thing as many bindings of a coloured Petri net transition.

The same behaviour can often be expressed as one transition or as many alternative transitions between the same states. That is, the transitions (s, R_1, s') and (s, R_2, s') yield together the same behaviour as the transition $(s, (R_1 \vee R_2), s')$. For instance, the bottommost transition of BANK could be replaced by two transitions labelled $[z < b] \mapsto \mathsf{Sy!}z$ and $[z = b] \mapsto \mathsf{Sy!}b$. It is not significant whether some behaviour is represented by one or more transitions.

We say that a data manipulation relation R is *empty* if and only if $R = \emptyset$, that is, $R \Leftrightarrow \mathsf{False}$. A transition with an empty R is never enabled. It could as well be removed from the state machine.

When expressing data manipulation relations mathematically using variable names, there is a problem: the previous and the next state may have a variable with the same name. For instance, each state of BANK has a variable called b. Therefore, to make it explicit whether we mean a $\bar{v}$-variable or a $\bar{w}$-variable, we write : after the name of the latter. The effect of the topmost transition of BANK on b can thus be written as $b - z = b$:, or, equivalently, $b{:} = b - z$. Considering : as a separate token we can use spaces differently and write $b := b - z$. This looks familiar and has the expected meaning. This is why we chose : as the "afterwards" specifier. In many other notations, the "afterwards" specifier is $'$, like in $b' = b - z$.

It is common that a variable inherits its value from a variable with the same name in the previous state. This could be expressed as $x{:} = x$, or as $x := x$. However, having to write and read a lot of that would be clumsy. Therefore, the ?-, etc., notation has an implicit assumption that if the value of a variable is not specified with ? or :=, and if also the previous state has a variable with the same name, then the variable gets the value of its namesake.

We are ready to present the formal definition.

Definition 1. *A* state machine *is a tuple* $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$*, where* S *is a set;* Θ *is a set of nonempty sets;* $\mathcal{T}$ *is a function from* S *to* $\Theta^{\times}$*;* $\hat{s} \in S$*;* $\hat{v} \in \mathcal{T}(\hat{s})$*;* Σ *is a set such that* $\tau \notin \Sigma$*; and* Δ *is a set of tuples of the form* (s, R, s')*, where* $s \in S$*,* $s' \in S$*, and* $R \subseteq \mathcal{T}(s) \times (\Sigma \cup \{\tau\}) \times \mathcal{T}(s')$*.*

In the sequel, we will have to talk about the *reachable part* of a state machine. It means the state machine induced by the states and transitions to which there are paths from the initial state. It is obvious that only the reachable part is relevant for the behaviour of the state machine. However, the reachable part is only an upper approximation to the relevant part, because, for instance, the guard of some transition may be equivalent to False. The precise relevant part is not necessarily easy to recognize. For instance, adding the guard $[z = b + 1]\!\!\rightarrow$ to the rightmost transition of BANK would make the top right corner state of BANK irrelevant, but that is not immediately obvious.

Definition 2. *The* reachable part *of a state machine* $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ *is the state machine* $(S', \Theta, \mathcal{T}', \Sigma, \Delta', \hat{s}, \hat{v})$*, where* $\mathcal{T}'$ *is the restriction of* $\mathcal{T}$ *to* S'*, and* S' *and* Δ' *are the smallest sets such that*

- $\hat{s} \in S'$*, and*
- *if* $s \in S'$ *and* $(s, R, s') \in \Delta$*, then* $s' \in S'$ *and* $(s, R, s') \in \Delta'$*.*

We will see in later sections that data manipulation relations are handy for the development of the theory. They liberate the concurrency part of the theory almost completely from concrete syntax and similar issues. They are usually naturally obtained from inscriptions written in languages for sequential computation. Their idea is more or less implicitly present in many concurrency formalisms, including coloured Petri nets. The following observation is at least implicitly known by many researchers. It is worth emphasizing.

Data manipulation relations are a handy way of combining the language for sequential computation to the theory of the behaviour of concurrent systems.

2.5 State Propositions

Many concurrency formalisms (especially temporal logics) associate logical propositions to states, such as "light is on" or "execution is in a critical section". They are called *state propositions*. We chose not to have state propositions in our definition, again to avoid complexity. However, we will comment on them in a couple of places.

In the absence of variables, they could be defined by associating to the state machine a set Π of state propositions, and a function $val : S \to 2^{\Pi}$ so that $val(s)$ is the set of state propositions that hold on s. In the presence of variables the definition becomes more complicated, because also their values may affect the truth value of a state proposition.

3 Concrete Behaviour

In this section we define the behaviour of a single state machine and demonstrate that it is essentially a state machine without variables. We also introduce bisimilarity and justify that it is a good notion for two behaviours to be equivalent at a detailed level.

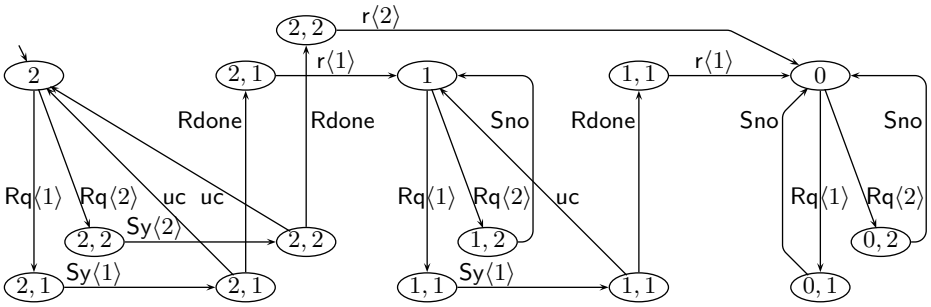


Fig. 4. BANK unfolded assuming that the types of b and z are $\{0, 1, 2\}$ and $\{1, 2\}$, and initially $b = 2$

3.1 Formal Definition of Behaviour: Unfolding

Mathematically, the behaviour of a state machine or system of state machines is represented as a *labelled transition system*, abbreviated *LTS*.

Definition 3. *A* labelled transition system *is a tuple* $(S, \Sigma, \Delta, \hat{s})$*, where* S *is a set,* Σ *is a set such that* $\tau \notin \Sigma$*,* $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$*, and* $\hat{s} \in S$*.*

The reachable part *of an LTS* $(S, \Sigma, \Delta, \hat{s})$ *is the LTS* $(S', \Sigma, \Delta', \hat{s})$*, where* S' *and* Δ' *are the smallest sets such that*

- $\hat{s} \in S'$*, and*
- *if* $s \in S'$ *and* $(s, a, s') \in \Delta$*, then* $s' \in S'$ *and* $(s, a, s') \in \Delta'$*.*

If Δ *is obvious from the context, then* $(s, a, s') \in \Delta$ *can also be written as* $s -a\rightarrow s'$*.*

The definition is thus otherwise the same as the definition of state machines, but there are no Θ, $\mathcal{T}$, and $\hat{v}$ components, and the elements of Δ have an a component instead of the R component. The components of the LTS have the same names as with state machines, that is, $\hat{s}$ is the initial state, and so on.

The behaviour of a state machine is obtained by *unfolding*. It resembles the unfolding of a coloured Petri net to a place/transition net, and similar operations are found also elsewhere in theoretical computer science. The states of the result could be represented as $(s, \bar{v})$, where s is a state of the state machine and $\bar{v}$ is the values of the variables of s. To avoid confusion with other tuple notation, we write $s\langle\bar{v}\rangle$ instead. We also treat s and $s\langle\rangle$ as synonyms. If the same s is encountered with different variable values $\bar{v}$ and $\bar{v}'$, then the result will have different states $s\langle\bar{v}\rangle$ and $s\langle\bar{v}'\rangle$. The initial state of the result is $\hat{s}\langle\hat{v}\rangle$. Only those $s\langle\bar{v}\rangle$ are included in the result that may be reached by executing the state machine starting at the initial state.

Figure 4 shows the behaviour of BANK with the types of variables replaced by so small sets that the figure can be drawn. To help reading, the states are

labelled with the $\bar{v}$ — that is, the values of b or b, z. For instance, the three end states of Rdone-transitions originate from the same state of BANK.

Let $s\langle\bar{v}\rangle$ be a state in the behaviour and (s, R, s') a transition of the state machine. It generates a transition from $s\langle\bar{v}\rangle$ to $s'\langle\bar{w}\rangle$ with action a to the behaviour if and only if $R(\bar{v}, a, \bar{w})$ holds. This transition is written as $(s\langle\bar{v}\rangle, a, s'\langle\bar{w}\rangle)$ or $s\langle\bar{v}\rangle -a\rightarrow s'\langle\bar{w}\rangle$. If there are event parameters and we want to show them, we write the transition as $(s\langle\bar{v}\rangle, a\langle\bar{p}\rangle, s'\langle\bar{w}\rangle)$ or $s\langle\bar{v}\rangle -a\langle\bar{p}\rangle\rightarrow s'\langle\bar{w}\rangle$, where $\bar{p}$ denotes the event parameters.

For instance, the Sy-transition of BANK is enabled only if $z \leq b$, so corresponding transitions start in Fig. 4 at those end states of Rq-transitions that are labelled with "$2, 1$", "$2, 2$", and "$1, 1$", but not at those labelled with "$1, 2$", "$0, 1$", and "$0, 2$". The transitions are labelled with $\mathsf{Sy}\langle 1\rangle$ and $\mathsf{Sy}\langle 2\rangle$.

Let us define the unfolding formally.

Definition 4. *Let $M = (S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ be a state machine. Its* behaviour *$\mathcal{B}(M)$ is the reachable part of the LTS $(S', \Sigma, \Delta', \hat{s}')$, where*

- $S' = \{\, s\langle\bar{v}\rangle \mid s \in S \wedge \bar{v} \in \mathcal{T}(s) \,\}$;
- $\Delta' = \{\, (s\langle\bar{v}\rangle, a, s'\langle\bar{w}\rangle) \mid \exists R : R(\bar{v}, a, \bar{w}) \wedge (s, R, s') \in \Delta \,\}$; *and*
- $\hat{s}' = \hat{s}\langle\hat{v}\rangle$.

It follows almost immediately from the definitions that if a state machine has no variables, it is essentially its own behaviour. The biggest difference is that while the state machine formalism allows to represent a set of transitions in one bunch as (s, R, s'), the LTS formalism requires to present each of the transitions individually. In the absence of variables we can let $\Theta = \emptyset$. For convenience, we denote also the void $\mathcal{T}$ and $\hat{v}$ with $\emptyset$.

Proposition 5. *Let $(S, \emptyset, \emptyset, \Sigma, \Delta, \hat{s}, \emptyset)$ be a state machine without variables. Its behaviour is isomorphic to the reachable part of $(S, \Sigma, \Delta', \hat{s})$, where $\Delta' = \{\, (s, a, s') \mid \exists R : R(a) \wedge (s, R, s') \in \Delta \,\}$.*

Proof. In the absence of variables, $\hat{v}$ and each $\bar{v}$ collapse to the empty list of values. The states of the behaviour are thus of the form $s\langle\rangle$, where s is a state of the state machine. In particular, $\hat{s}' = \hat{s}\langle\rangle$. The data manipulation relations only have a-components. Therefore, the transition (s, R, s') introduces precisely the transitions $(s\langle\rangle, a, s'\langle\rangle)$ to the behaviour, where $R(a)$ holds. Finally, also the definition of behaviour has restriction to the reachable part. □

We also want to demonstrate that the behaviour of any state machine (possibly with variables) is essentially a state machine without variables. By definition, it is an LTS. It suffices to show that for each LTS, there is a state machine without variables whose behaviour is isomorphic to the reachable part of the LTS. We do that next. In the construction, a separate state machine transition is made from each LTS transition. The data manipulation relation of that transition is $\{a\}$, that is, the relation R such that $R(a)$ holds and $R(x)$ does not hold if $x \neq a$. The correctness of the proposition is obvious.

Proposition 6. *Let* $(S, \Sigma, \Delta, \hat{s})$ *be an LTS. Its reachable part is isomorphic to the behaviour of the state machine* $(S, \emptyset, \emptyset, \Sigma, \Delta', \hat{s}, \emptyset)$*, where* $\Delta' = \{ (s, \{a\}, s') \mid (s, a, s') \in \Delta \}$.

So the behaviours of state machines can be treated as state machines. This unification of state machines with their behaviours gives the theory a lot of power that we will enjoy in later sections.

> The behaviour of a state machine is essentially a state machine without variables, and the behaviour of a state machine without variables is essentially the state machine itself.

3.2 Equivalence of Detailed Behaviours

From the point of view of externally observable behaviour, labels of events (i.e., actions) are important, but names of states are not. If one wants to look at properties of states during verification, then one must either use the state propositions of Sect. 2.5, or reason the necessary properties from visible events. One may, for instance, introduce the actions enter and leave to verify that two state machines are not in their critical sections at the same time. We already utilised the insignificance of state names in the previous subsection, by considering two LTSs essentially the same if there is an isomorphism between their states.

However, isomorphism often fails to unify intuitively obvious instances of "same behaviour". For instance, consider Fig. 4. It seems clear that fusing the two states at bottom right corner (the start states of Sno-transitions) does not change the externally observable behaviour. Isomorphism cannot reflect that, because isomorphic LTSs always have the same number of states, and state fusion changes the number of states.

Bisimilarity is an equivalence notion that is much better than isomorphism in unifying intuitively equivalent behaviours, while it still avoids unifying behaviours when it should not unify. It is strictly weaker than isomorphism, that is, isomorphic LTSs are always bisimilar, but bisimilar LTSs are not always isomorphic. Many researchers consider bisimilarity as *the* notion of "same behaviour at the detailed level". On the other hand, from the point of view of externally observable behaviour, the detailed level is much less important than the abstract level that we will discuss in Sect. 5. Therefore, bisimilarity is not a goal in itself but only a useful technical tool on the way to the real goal. However, it is a tool that one must master to understand concurrency.

Bisimilarity is sometimes called *strong bisimilarity*, to distinguish it from "weak bisimilarity" that we will meet in Sect. 5.6.

Bisimilarity is defined between states. It could be defined between the sets of states of two LTSs, but it is more flexible to define it between the states of a single LTS, and apply it to two LTSs by first combining them into one LTS by taking their disjoint union. Intuitively, taking the disjoint union simply means drawing the two LTSs on the same sheet of paper and pretending that they are two isolated regions of the same LTS (whose initial state is chosen

arbitrarily). The formal definition of disjoint union is well-known and we skip it. Two LTSs are bisimilar if and only if their initial states are and they have the same alphabet.

To discuss the basic idea of bisimilarity, consider two bisimilar states s_1 and s_2. An essential phenomenon in concurrency is *nondeterminism*, that is, a state may have more than one output transition with the same action. The goal of the definition of bisimilarity is to guarantee that whatever output transitions a state has, the bisimilar state is able to simulate. That is, for each output transition $s_1 - a \rightarrow s_1'$ of s_1, s_2 must have an output transition with the same action a and, furthermore, the futures after the original and simulating transitions must be somehow the same. Let $s_2 - a \rightarrow s_2'$ be that transition. The equivalence of futures is ensured by building the definition so that also s_1' and s_2' will be bisimilar. Of course, it is also required that whatever output transitions s_2 has, s_1 must be able to simulate. However, it is not required that the mapping between transitions and their simulating transitions is one-to-one; that is, a transition may simulate and be simulated by many transitions.

The description above is not precise enough to qualify as a definition of bisimilarity. For instance, the description would allow us to declare that s_1 and s_2 are bisimilar if and only if $s_1 = s_2$, which would obviously be against our goal. Therefore, the formal definition of bisimilarity relies on the auxiliary notion of *bisimulation*. Bisimulation is any relation on states that satisfies the above description. It can be proven that there is a unique weakest bisimulation (it is the union of all bisimulations). This unique weakest bisimulation is the bisimilarity.

Definition 7. *Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS. The relation "$\sim$" $\subseteq S \times S$ is a* bisimulation*, if and only if for every $s_1 \in S$, $s_2 \in S$, $s \in S$, and $a \in \Sigma \cup \{\tau\}$ such that $s_1 \sim s_2$ the following hold:*

- *If $s_1 - a \rightarrow s$, then there is $s' \in S$ such that $s_2 - a \rightarrow s'$ and $s \sim s'$.*
- *If $s_2 - a \rightarrow s$, then there is $s' \in S$ such that $s_1 - a \rightarrow s'$ and $s' \sim s$.*

We say that $s_1 \in S$ and $s_2 \in S$ are bisimilar*, if and only if there is a bisimulation "$\sim$" such that $s_1 \sim s_2$.*

We say that the LTSs $(S_1, \Sigma_1, \Delta_1, \hat{s}_1)$ and $(S_2, \Sigma_2, \Delta_2, \hat{s}_2)$ are bisimilar *if and only if $\Sigma_1 = \Sigma_2$, and $\hat{s}_1$ and $\hat{s}_2$ are bisimilar in the disjoint union of the LTSs.*

The definition of the bisimilarity of LTSs requires that their alphabets must be the same. This is because the alphabet determines whether the state machine may prevent other state machines from executing transitions, as was discussed in Sect. 2.2. It is thus essential from the point of view of neighbour LTSs.

If the formalism is extended with multiple initial states, then the condition that $\hat{s}_1$ and $\hat{s}_2$ are bisimilar must be replaced with the following condition: every initial state of the first LTS has a bisimilar initial state in the second LTS, and vice versa. If the formalism is extended with state propositions, then the requirement must be added that bisimilar states give the same truth values to state propositions. For simplicity, we will not take these extensions into account in the subsequent discussion.

The following result could be easily proven with induction. It says that the notion of simulation of single transitions given by the definition of bisimilarity extends to arbitrary sequences of transitions.

Proposition 8. *Let "$\sim$" denote bisimilarity. If $s_0 -a_1\to s_1 -a_2\to \cdots -a_n\to s_n$ and $s_0 \sim s'_0$, then there are s'_1, s'_2, ..., s'_n such that $s_1 \sim s'_1$, $s_2 \sim s'_2$, ..., $s_n \sim s'_n$ and $s'_0 -a_1\to s'_1 -a_2\to \cdots -a_n\to s'_n$. A similar result holds for infinite sequences $s_0 -a_1\to s_1 -a_2\to \cdots$.*

It is not difficult to prove that bisimilarity indeed is an equivalence. To prove that two states are bisimilar, it suffices to present a bisimulation that relates them. Bisimulations are not necessarily equivalences. In particular, the empty relation (the one where $s_1 \sim s_2$ never holds, no matter what s_1 and s_2 are) is a bisimulation. When checking whether two LTSs are bisimilar, only the reachable parts matter, because, so to speak, the empty relation can be used elsewhere. The empty relation cannot be used in the reachable parts because of the requirement that $\hat{s}_1 \sim \hat{s}_2$. From this, Proposition 8 implies that the relation must be nonempty throughout the reachable parts.

For any given finite LTS, there is a unique (modulo isomorphism) smallest bisimilar LTS. It can be found by leaving out the unreachable parts and fusing equivalent states in the reachable part. This can be done (at least in a mathematical sense) also to infinite LTSs, but the result cannot be called "smallest", because it is not a well-defined concept with infinite objects. The fusion operation is defined below. The Δ'-part of the definition goes through all transitions in Δ, but it would suffice to take one state in each $[[s]]$ and go through just their output transitions. This is because the definition of bisimilarity guarantees that if one state in an equivalence class has an a-transition to an equivalence class, then every state in the former equivalence class has an a-transition to the latter equivalence class.

Definition 9. *Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS, and let "$\sim$" denote bisimilarity. Its* quotient *modulo bisimilarity is the LTS $(S', \Sigma, \Delta', \hat{s}')$, where*

- $[[s]] = \{\, s' \mid s \sim s' \,\}$,
- $S' = \{\, [[s]] \mid s \in S \,\}$,
- $\Delta' = \{\, ([[s]], a, [[s']]) \mid (s, a, s') \in \Delta \,\}$, *and*
- $\hat{s}' = [[\hat{s}]]$.

In the case of finite LTSs, the above construction can be done in $O(|S| + |\Delta| \log |S|)$ time [35]. (The algorithm in [8] has been influential but does not meet this time bound.) This is fast enough for almost all practical purposes. The construction can also be used for checking whether two states or LTSs are bisimilar. This is a big difference from isomorphism, which is believed not to be checkable in worst-case polynomial time.

Bisimilarity is a most appropriate notion of "same behaviour" at the detailed level of behaviour.

4 Putting State Machines Together

In this section we discuss how a system is put together from state machines. We introduce three operators for that and then combine them into one flexible operator. We discuss how the behaviour of the system is determined as a function of the behaviours of its parts, and point out that the same behaviour can also be obtained by first putting the state machines together. Then we comment on the notions of input and output, and mention some other operators used in process algebras.

4.1 Parallel Composition

Many different parallel composition operators have been defined in the literature. The most suitable for our purpose can be called *alphabet-based synchronization*. We define it first for LTSs.

The operator works as we discussed towards the end of Sect. 2.2. Each state of the result is a tuple consisting of the states of the components. So the joint LTS keeps track of the states of the component LTSs. The joint LTS executes a transition labelled with τ when precisely one of the component LTSs executes a τ-transition. If $a \neq \tau$, the joint LTS executes an a-transition when precisely those component LTSs execute simultaneously an a-transition which have a in their alphabets. The components that do not participate the execution stay in their current states. The result is restricted to the reachable part, because the unreachable part is often big and, as we have pointed out, it is irrelevant for the behaviour.

Definition 10. *Let* $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$ *be LTSs for* $1 \leq i \leq n$. *Their* parallel composition $L_1 \,||\, \cdots \,||\, L_n$ *is the reachable part of the LTS* $(S, \Sigma, \Delta, \hat{s})$, *where*

- $S = S_1 \times \cdots \times S_n$;
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$;
- *if* $(s_1, \ldots, s_n) \in S$ *and there is* $1 \leq j \leq n$ *such that for every* $1 \leq i \leq n$
  - *either* $i = j$ *and* $(s_i, \tau, s'_i) \in \Delta_i$
  - *or* $i \neq j$ *and* $s'_i = s_i$,

  then $((s_1, \ldots, s_n), \tau, (s'_1, \ldots, s'_n)) \in \Delta$;
- *if* $a \in \Sigma$, $(s_1, \ldots, s_n) \in S$, *and for every* $1 \leq i \leq n$
  - *either* $a \in \Sigma_i$ *and* $(s_i, a, s'_i) \in \Delta_i$
  - *or* $a \notin \Sigma_i$ *and* $s'_i = s_i$,

  then $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$;
- Δ *has no other elements; and*
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$.

If the ordering of the components is changed (e.g., from $L_1 \,||\, L_2$ to $L_2 \,||\, L_1$), the names of states change accordingly (from (s_1, s_2) to (s_2, s_1)), but the result is isomorphic to the original. Therefore, it is appropriate to say that $||$ is commutative. It is also associative, because $(L_1 \,||\, L_2) \,||\, L_3$ differs from $L_1 \,||\, (L_2 \,||\, L_3)$ and

even from $L_1 \,||\, L_2 \,||\, L_3$ only by the names of states: $((s_1, s_2), s_3)$ vs. $(s_1, (s_2, s_3))$ vs. (s_1, s_2, s_3).

With $||$, the alphabets determine synchronization. A widely used alternative is to list the synchronizing actions in the operator, e.g., $L_1 \,|[a, b]|\, L_2$. In this example, if $c \neq a$ and $c \neq b$, then c-transitions of L_1 and L_2 do not synchronize, even if both L_1 and L_2 can execute them. Obviously $L_1 \,||\, L_2$ is obtained as $L_1 \,|[a_1, \ldots, a_n]|\, L_2$, where $\{a_1, \ldots, a_n\}$ is the intersection of the alphabets of L_1 and L_2. On the other hand, $|[\cdots]|$ can be easily built from $||$ and the renaming operator of Sect. 4.3. So, from the point of view of expressivity, it does not matter which one we choose. However, $|[\cdots]|$ is not associative, which makes its mathematics more complicated and may also confuse users.

The widely used CCS parallel composition operator $|$ does not allow more than two components to synchronize to the same event. (CCS is *Calculus of Communicating Systems* [20].) It is a significant disadvantage compared to $||$. Using the ideas in Sect. 4.4, $|$ can be easily constructed from $||$ and the operators in Sect. 4.2 and 4.3. On the other hand, it is far from obvious how to construct $||$ from $|$ and the other operators of CCS and this tutorial. There is also a complexity-theoretic sense [36] in which $||$ is simpler than $|[\cdots]|$ and $|$.

Bisimilarity is a *congruence* with respect to $||$. That is, if L_i and L'_i are bisimilar for $1 \leq i \leq n$, then $L_1 \,||\, \cdots \,||\, L_n$ is bisimilar to $L'_1 \,||\, \cdots \,||\, L'_n$. The importance of the congruence property will be discussed in Sect. 5.2. Bisimilarity is a congruence also with respect to the variants that we briefly discussed above, the operators that will be discussed in the remainder of this section, and, indeed, with respect to almost all operators that have been suggested in the process algebra literature.

Let M be a state machine and $\mathcal{B}(M)$ its behaviour. We could now define the behaviour of a parallel composition of state machines as the parallel composition of the behaviours of the state machines: $\mathcal{B}(M_1) \,||\, \cdots \,||\, \mathcal{B}(M_n)$. However, computing the behaviours is often very expensive, as it involves unfolding. So we would like to be able to compute a state machine $M_1 \,||\, \cdots \,||\, M_n$ such that $\mathcal{B}(M_1 \,||\, \cdots \,||\, M_n) = \mathcal{B}(M_1) \,||\, \cdots \,||\, \mathcal{B}(M_n)$. To do that, we must first discuss the joint effect of data manipulation relations of parallel state machines.

For example, assume that there are three state machines, and they have transitions labelled with $[n > 0] \rightarrow \mathsf{a}!n?n$, $[i \leq 3] \rightarrow \mathsf{a}!i?j$, and $\mathsf{a}?k!0$, respectively. The inscriptions in the first parameter position imply that i and n must have the same value, and that value will be stored into k. It is also the value of the first event parameter, which we denote with p_1. The guards imply that the value must be 1, 2, or 3. The inscriptions in the second parameter position imply that the value of the second event parameter p_2 is 0, and n and j will be 0. Assuming that there are no other variables, the individual data manipulation relations are $n = p_1 > 0 \wedge n := p_2$, $i = p_1 \leq 3 \wedge i := i \wedge j := p_2$, and $k := p_1 \wedge p_2 = 0$. The joint relation is $0 < n = i = p_1 \leq 3 \wedge p_2 = 0 \wedge n := 0 \wedge i := i \wedge j := 0 \wedge k := n$.

When formalizing the joint effect, we must make it precise how variables and actions are treated in the conjunction of data manipulation relations. Variables of different state machines must be treated as different variables even if they have

the same name, but actions must be shared. Not necessarily all state machines participate in a transition. We must specify that those who do not, do not change the values of their variables. For that purpose, for each list of variables we define the *identity relation* $I(\bar{v}, a, \bar{w}) \Leftrightarrow \bar{w} = \bar{v}$, that is, the variable values stay the same and the action does not matter. By $(R_1 \wedge \cdots \wedge R_n)(a)$ we mean the relation $R(\bar{v}_1, \ldots, \bar{v}_n, a, \bar{w}_1, \ldots, \bar{w}_n) \Leftrightarrow R_1(\bar{v}_1, a, \bar{w}_1) \wedge \cdots \wedge R_n(\bar{v}_n, a, \bar{w}_n)$.

We can now define $M_1 \,||\, \cdots \,||\, M_n$.

Definition 11. *Let* $M_i = (S_i, \Theta_i, \mathcal{T}_i, \Sigma_i, \Delta_i, \hat{s}_i, \hat{v}_i)$ *be state machines for* $1 \leq i \leq n$. *Their* parallel composition $M_1 \,||\, \cdots \,||\, M_n$ *is the reachable part of the state machine* $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$, *where*

- $S = S_1 \times \cdots \times S_n$;
- $\Theta = \Theta_1 \cup \cdots \cup \Theta_n$;
- $\mathcal{T}(s) = \mathcal{T}_1(s_1) \times \cdots \times \mathcal{T}_n(s_n)$ *for every* $s = (s_1, \ldots, s_n) \in S$;
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$;
- I_i *is the identity relation for the variables of* s_i;
- *if* $(s_1, \ldots, s_n) \in S$ *and there is* $1 \leq j \leq n$ *such that for every* $1 \leq i \leq n$
  - *either* $i = j$, $(s_i, R_i, s'_i) \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, \tau, \bar{w}_i)$
  - *or* $i \neq j$, $R_i \Leftrightarrow I_i$, *and* $s'_i = s_i$,

  then $((s_1, \ldots, s_n), (R_1 \wedge \cdots \wedge R_n)(\tau), (s'_1, \ldots, s'_n)) \in \Delta$;
- *if* $a \in \Sigma$, $(s_1, \ldots, s_n) \in S$, *and for every* $1 \leq i \leq n$
  - *either* $a \in \Sigma_i$, $(s_i, R_i, s'_i) \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, a, \bar{w}_i)$
  - *or* $a \notin \Sigma_i$, $R_i \Leftrightarrow I_i$, *and* $s'_i = s_i$,

  then $((s_1, \ldots, s_n), (R_1 \wedge \cdots \wedge R_n)(a), (s'_1, \ldots, s'_n)) \in \Delta$;
- Δ *has no other elements;*
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$; *and*
- $\hat{v} = (\hat{v}_1, \ldots, \hat{v}_n)$.

The purpose of the conditions $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, \tau, \bar{w}_i)$ and $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, a, \bar{w}_i)$ is to avoid creating transitions with obviously empty data manipulation relations. Without them, the definition would, for instance, create a dead τ-transition from the ci-transition of ATM. If the conditions are too difficult to check precisely in a practical situation, then one may use an upper approximation and accept that the result may have extra dead transitions. For instance, one may generate all transitions where the gate names match.

Figure 5 shows the parallel composition of a modified ATM and BANK. To get a readable figure, the channels have been left out, and ATM and BANK interact directly. Losses of messages in the channels are modelled by additional transitions in ATM and BANK that read the incoming message but do not change the state nor the values of local variables. To keep the figure readable, the balance variable b has been removed.

The horizontal and vertical arrows correspond to transitions that either ATM or BANK executes alone, and so does the arrow labelled with nm. There are only four transitions in which both participate simultaneously. In two of them, labelled q!x $z := x$ and y!z $x := z$, data is passed, which is shown by the assignment in the inscription of the arrow. The data manipulation relations of

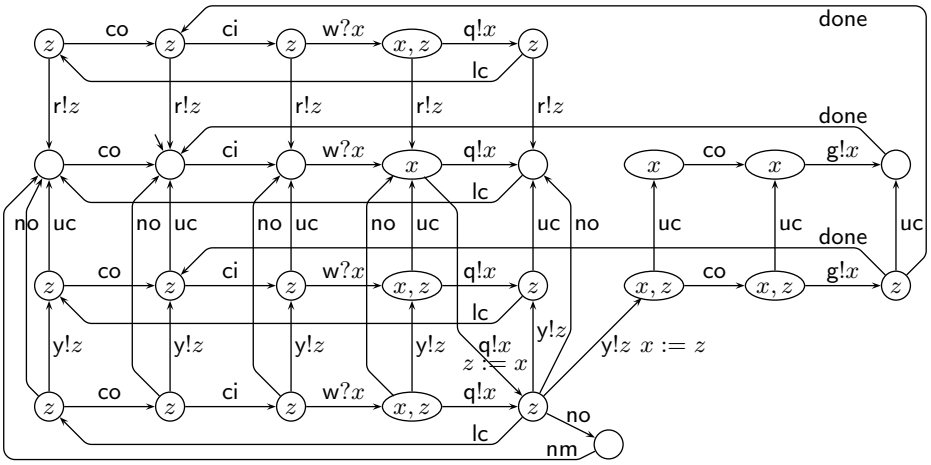


Fig. 5. The parallel composition of ATM and BANK with variable b removed and channels replaced by direct synchronization that can lose messages

the synchronizing q-transitions are $p_1 = x$ and $z := p_1$. Their conjunction is equivalent to $p_1 = x \wedge z := x$. In the figure, this is represented by $!x$ and $z := x$.

We are ready to show that Definition 11 produces what it should.

Proposition 12. *$\mathcal{B}(M_1 \,||\, \cdots \,||\, M_n)$ and $\mathcal{B}(M_1) \,||\, \cdots \,||\, \mathcal{B}(M_n)$ are isomorphic.*

Proof. The states of $\mathcal{B}(M_1 \,||\, \cdots \,||\, M_n)$ are of the form $(s_1, \ldots, s_n)\langle \bar{v}_1, \ldots, \bar{v}_n \rangle$, while the states of $\mathcal{B}(M_1) \,||\, \cdots \,||\, \mathcal{B}(M_n)$ are of the form $(s_1\langle \bar{v}_1 \rangle, \ldots, s_n\langle \bar{v}_n \rangle)$. In both ways of computing the result, τ implies that precisely one state machine participates. In both ways, if $a \neq \tau$, each a-transition is participated by precisely those state machines which have a in their alphabets. In both ways, the state machines that do not participate keep their states and variable values. In both ways, the state machines that do participate change their states similarly, and change their variable values according to their data manipulation relations. In both ways, the result is restricted to the reachable part. □

Of course, it would be possible to define an operation that both puts the state machines together and unfolds the result. Such an operation corresponds to traditional state space construction. It is an advantage of process algebras that the behaviour need not be constructed in one big step but it can be constructed in many sub-steps, and there is freedom in the order of the sub-steps. We will see in Sect. 5.8 that useful things can be done between the sub-steps. It would not be possible, if the behaviour had to be computed in one big batch.

> The behaviour of a parallel composition is the parallel composition of the behaviours of its components. It can be computed in one big batch or divided to parallel composition and unfolding steps in many different ways.

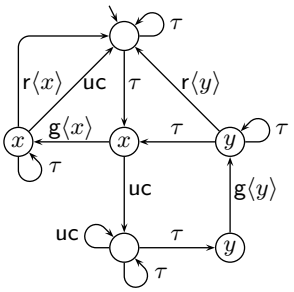


Fig. 6. The user's money view to the simplified cash dispenser system in Fig. 5

4.2 Hiding

The hiding operator converts visible actions to invisible. With it one can choose a view to a system. The behaviour of the system can be projected to the chosen view with techniques discussed in Sect. 5.

Figure 6 shows the projection of Fig. 5 to the view of the user's money. The view was chosen by leaving g, r, and uc visible, and hiding everything else. The figure has been produced manually but, excluding the variables x and y, it is the same as an automatically generated figure with the data type restricted to a singleton set. Most details of the figure cannot be explained before discussing the theory in Sect. 5.4, but some observations can be made already now. For instance, every "reduce balance" transition (r) is preceded by a "give money" transition (g) with the same amount. So the system never charges the balance without giving the money.

If the channels in Fig. 3 are used, then new phenomena emerge. For instance, the system can charge a wrong amount of money from the account! The following sequence of events leads to the error. The user tries to withdraw x units of money. The transaction progresses successfully up to $\mathsf{Sy}\langle x\rangle$, but then both ATM and BANK give up and return to their initial states. While doing so, BANK executes uc. The user tries again with a new amount y. After executing $\mathsf{Sq}\langle y\rangle$, ATM gets the "yes"-answer that was left over in CH2 from the previous attempt, and gives the user x units of money with $\mathsf{g}\langle x\rangle$. BANK reads the second query by executing $\mathsf{Rq}\langle y\rangle$ and replies "yes" to it. ATM sends "done". BANK receives it and reduces y units of money from the account.

Instead of fixing the cash dispenser system, we continue discussing the theory. The definition of the hiding operator on LTSs is simple. The hidden actions are removed from the alphabet, because they become internal to the LTS. This makes it possible to use their names as action names in other parts of the system.

Definition 13. *Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS, and A be a set. The result of* hiding *A in L is the LTS $L \setminus A = (S, \Sigma', \Delta', \hat{s})$, where*

- $\Sigma' = \Sigma \setminus A$;
- *if $(s, a, s') \in \Delta$ and $a \notin A$, then $(s, a, s') \in \Delta'$;*

– *if* $(s, a, s') \in \Delta$ *and* $a \in A$, *then* $(s, \tau, s') \in \Delta'$; *and*
– Δ' *has no other elements.*

It might seem natural to require that $A \subseteq \Sigma$. However, extra elements in A do not affect the result (even $\tau \in A$ is harmless), and the operator is easier to use if one need not ensure that A indeed is a subset of Σ. So we do not make the requirement.

In the definition of the hiding operator of state machines, hiding must be defined for the data manipulation relations, because the same relation may induce transitions with both hidden and unhidden actions. The first part of the definition of $R \setminus A$ keeps transitions with unhidden actions, and the second part generates τ-transitions from transitions with hidden actions.

Definition 14. *Let* $M = (S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ *be a state machine, and* A *be a set. The result of* hiding A *in* M *is the state machine* $M \setminus A = (S, \Theta, \mathcal{T}, \Sigma', \Delta', \hat{s}, \hat{v})$, *where*

– $\Sigma' = \Sigma \setminus A$;
– $(R \setminus A)(\bar{v}, a, \bar{w}) \Leftrightarrow a \notin A \wedge R(\bar{v}, a, \bar{w}) \vee a = \tau \wedge \exists b \in A : R(\bar{v}, b, \bar{w})$; *and*
– $\Delta' = \{\, (s, R \setminus A, s') \mid (s, R, s') \in \Delta \,\}$.

Also hiding has the property that it does not matter whether it is done before or after unfolding, that is, $\mathcal{B}(M \setminus A) = \mathcal{B}(M) \setminus A$. (This time "=" is equality and not just isomorphism.)

A number of properties obviously hold, such as

– $(M \setminus A) \setminus B = M \setminus (A \cup B)$.
– If $A \cap \Sigma_2 = \emptyset$, then $(M_1 \setminus A) \,||\, M_2 = (M_1 \,||\, M_2) \setminus A$.

4.3 Relational Renaming

Renaming means changing gate names or actions. As such, it makes it possible to specify a state machine once and use it in more than one place. For instance, one may specify the dining philosophers' system by specifying a single dining philosopher with actions take_left, take_right, release_left, and release_right and a single chop stick with actions take and release, and taking several copies of them, renaming the actions to take_1, take_2, and so on.

Simple renaming and || suffice for the philosophers' system, but they run into trouble in the following kind of a situation. There are many servers and even more clients. Any server can serve any client. When a client needs service, it sends a general call that precisely one free server synchronizes to, but that server can be any of the free servers. If no server is free, the call transition is blocked. We want the outside world to see which client and server synchronized.

This situation can be modelled with a more general form of renaming, where one may map a single action to more than one action. It is called *multiple renaming* or *relational renaming*. We skip the definition for LTSs and only show the definition for state machines.

Definition 15. *Let $M = (S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ be a state machine. A* renaming relation *for M is any set Φ of pairs such that the domain of Φ is precisely Σ, that is, $\{\ a \mid \exists b : (a,b) \in \Phi\ \} = \Sigma$, and τ is not in the range of Φ, that is, $\neg\exists a : (a,\tau) \in \Phi$. The result of applying Φ to M is the state machine $M\Phi = (S, \Theta, \mathcal{T}, \Sigma', \Delta', \hat{s}, \hat{v})$, where*

- $\Sigma' = \{\ b \mid \exists a : (a,b) \in \Phi\ \}$;
- $(R\Phi)(\bar{v}, b, \bar{w}) \Leftrightarrow \exists a : (a,b) \in \Phi \wedge R(\bar{v}, a, \bar{w}) \vee b = \tau \wedge R(\bar{v}, \tau, \bar{w})$; *and*
- $\Delta' = \{\ (s, R\Phi, s') \mid (s, R, s') \in \Delta\ \}$.

The purpose of the restriction on the domain of Φ is to simplify the theory by ruling out unnecessary special cases. Without it, one could remove some transitions altogether by leaving their action a without a pair $(a,b) \in \Phi$; and one could add extra members to Σ' by having $(a,b) \in \Phi$ such that $a \notin \Sigma$. The restriction does not imply loss of generality, because one can have the same effects with $||$. Let $\mathbf{stop}_A$ be the state machine with one state, no variables, no transitions, and the alphabet A. One can remove the a-transitions of M for every $a \in A$ by writing $(M \,||\, \mathbf{stop}_A) \setminus A$. One can add B to the alphabet by writing $M \,||\, \mathbf{stop}_{B \setminus \Sigma}$.

The client–server example can now be modelled as

$$C\Phi_1 \,||\, \cdots \,||\, C\Phi_n \,||\, S\Phi'_1 \,||\, \cdots \,||\, S\Phi'_m \ ,$$

where

- C runs in a loop $s_1 -\mathsf{call}\rightarrow s_2 -\mathsf{reply}\rightarrow s_1$;
- S runs in a loop $s_1 -\mathsf{call}?i\rightarrow s_2\langle i\rangle -\mathsf{reply}!i\rightarrow s_1$;
- $\Phi_{i,j} = \{\ (\mathsf{call}, \mathsf{call}\langle i,j\rangle), (\mathsf{reply}, \mathsf{reply}\langle i,j\rangle)\ \}$;
- $\Phi_i = \Phi_{i,1} \cup \cdots \cup \Phi_{i,m}$;
- $\Phi'_{i,j} = \{\ (\mathsf{call}\langle i\rangle, \mathsf{call}\langle i,j\rangle), (\mathsf{reply}\langle i\rangle, \mathsf{reply}\langle i,j\rangle)\ \}$; and
- $\Phi'_j = \Phi'_{1,j} \cup \cdots \cup \Phi'_{n,j}$.

The variable i in the server makes it reply to the right client. Φ_i adds the identity i of the client to its actions. It also takes one copy of each action for each server, so that the client can synchronize with any server. Φ'_j adds the identity of the server to its actions.

4.4 Synchronization Rules

With the operators introduced this far, one can write complicated expressions such as $((M_1\Phi_1 \,||\, M_2\Phi_2) \setminus A) \,||\, M_3$. However, intuitively each transition of any such system consists of some state machines participating via some actions, other state machines not participating, and the result having some action. The resulting action may be τ even if none of the original actions is, but if any of the original actions is τ, then only that state machine participates, and the resulting action is τ.

We now make this idea precise. Let - be a symbol that is not in any alphabet. It will denote that the state machine does not participate the transition. (We could have used τ for that purpose but felt it confusing, because not participating is not the same thing as participating via a τ-transition.)

Definition 16. *Let* $M_i = (S_i, \Theta_i, \mathcal{T}_i, \Sigma_i, \Delta_i, \hat{s}_i, \hat{v}_i)$ *be state machines for* $1 \leq i \leq n$. *Let* Σ *be a set such that* $\tau \notin \Sigma$. *A* synchronization rule *for them is a tuple* $r = (a_1, \ldots, a_n; a)$, *where* $a \in \Sigma \cup \{\tau\}$ *and* $a_i \in \Sigma_i \cup \{\text{-}\}$ *for* $1 \leq i \leq n$, *and at least one* $a_i \neq$ -. *We let* $r[0] = a$ *and* $r[i] = a_i$ *for* $1 \leq i \leq n$.

Let $\mathcal{Y}$ *be a set of synchronization rules for* M_1, ..., M_n, *and* Σ. *Then* $\mathcal{Y}(M_1, \ldots, M_n)$ *is the reachable part of the state machine* $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$, *where*

- $S = S_1 \times \cdots \times S_n$;
- $\Theta = \Theta_1 \cup \cdots \cup \Theta_n$;
- $\mathcal{T}(s) = \mathcal{T}_1(s_1) \times \cdots \times \mathcal{T}_n(s_n)$ *for every* $s = (s_1, \ldots, s_n) \in S$;
- I_i *is the identity relation for the variables of* s_i;
- *if* $(s_1, \ldots, s_n) \in S$ *and there is* $1 \leq j \leq n$ *such that for every* $1 \leq i \leq n$
  - *either* $i = j$, $(s_i, R_i, s_i') \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, \tau, \bar{w}_i)$
  - *or* $i \neq j$, $R_i \Leftrightarrow I_i$, *and* $s_i' = s_i$,

  then $((s_1, \ldots, s_n), (R_1 \wedge \cdots \wedge R_n)(\tau), (s_1', \ldots, s_n')) \in \Delta$;
- *if* $(a_1, \ldots, a_n; a) \in \mathcal{Y}$, $(s_1, \ldots, s_n) \in S$, *and for every* $1 \leq i \leq n$
  - *either* $a_i \in \Sigma_i$, $(s_i, R_i, s_i') \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, a_i, \bar{w}_i)$
  - *or* $a_i =$ -, $R_i \Leftrightarrow I_i$, *and* $s_i' = s_i$,

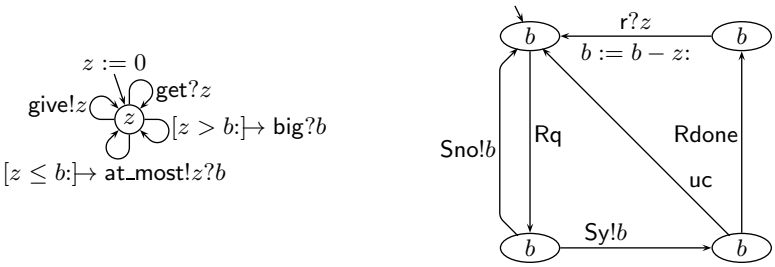
  then $((s_1, \ldots, s_n), R, (s_1', \ldots, s_n')) \in \Delta$, *where* $R(\bar{v}_1, \ldots, \bar{v}_n, a, \bar{w}_1, \ldots, \bar{w}_n) \Leftrightarrow R_1(\bar{v}_1, a_1, \bar{w}_1) \wedge \cdots \wedge R(\bar{v}_n, a_n, \bar{w}_n)$;
- Δ *has no other elements;*
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$; *and*
- $\hat{v} = (\hat{v}_1, \ldots, \hat{v}_n)$.

This definition is not much more difficult to understand than the previous ones, and it makes it possible to specify arbitrary synchronization patterns. Most, or perhaps all, major parallel composition operators in process algebras can be constructed with synchronization rules.

Synchronization rules can also be used to represent local variables as parallel state machines. For instance, Fig. 7 shows how the variable z of BANK could be replaced by a state machine that is synchronized with the rest of BANK. In this example, b of BANK-MAIN is kept as a local variable. $z := 0$ specifies the initial value of z. b: refers to the afterwards value of b. So, e.g., $[z > b{:}] \mapsto$ big?b means that big$\langle j \rangle$ is available with those values of j that satisfy $z > j$. The third rule and the inscriptions of the transitions say that at_most has two event parameters, the first carrying the current value of z and the second carrying the same or bigger value; this latter value is also the event parameter of Sy of BANK-MAIN and thus equal to b; and the outside world sees Sy with the value of z.

This implementation of BANK is not isomorphic to Fig. 2. The difference is that while BANK-MAIN is in its initial state, z has no value in Fig. 2, but in Fig. 7 it keeps its previous value. This difference does not affect the behaviour significantly, because the value of z in the initial state is not used and is overwritten by the next transition. Indeed, the two models of BANK are bisimilar.

The example has a synchronization rule for each possible combination of event parameter values for each gate. There are thus infinitely many rules. This is not



(	get$\langle i \rangle$	, Rq	; Rq$\langle i \rangle$	)	for $i \in \mathbb{N}$
(	big$\langle j \rangle$	, Sno$\langle j \rangle$	; Sno	)	for $j \in \mathbb{N}$
(	at_most$\langle i, j \rangle$	, Sy$\langle j \rangle$	; Sy$\langle i \rangle$	)	for $i \in \mathbb{N}$ and $j \in \mathbb{N}$
(	-	, uc	; uc	)	
(	-	, Rdone	; Rdone	)	
(	give$\langle i \rangle$	, r$\langle i \rangle$	; r$\langle i \rangle$	)	for $i \in \mathbb{N}$

Fig. 7. BANK made of Z and BANK-MAIN state machines with synchronization rules

a problem for developing theoretical results. In a practical situation, one can use suitable notation for representing rules in bunches. Indeed, Fig. 7 has only six such bunches.

Analogously to earlier operators, we should prove that $\mathcal{B}(\mathcal{Y}(M_1, \ldots, M_n))$ is isomorphic to $\mathcal{Y}(\mathcal{B}(M_1), \ldots, \mathcal{B}(M_n))$. Instead of doing that directly, we will get the result for free from another result: the effect of synchronization rules can be built from the other operators discussed so far. This implies that synchronization rules can be considered shorthand notation, and it suffices to develop the theory for the other operators.

Proposition 17. *Let $M_i = (S_i, \Theta_i, \mathcal{T}_i, \Sigma_i, \Delta_i, \hat{s}_i, \hat{v}_i)$ be state machines for $1 \leq i \leq n$. Let Σ be a set such that $\tau \notin \Sigma$, and let $\mathcal{Y}$ be a set of synchronization rules for them. Then $\mathcal{Y}(M_1, \ldots, M_n) =$*

$$\Big(\Big(\big((M_1 \,||\, \mathbf{stop}_{A_1}) \setminus A_1\big)\Phi_1 \,||\, \cdots \,||\, \big((M_n \,||\, \mathbf{stop}_{A_n}) \setminus A_n\big)\Phi_n\Big) \setminus A\Big)\Phi \,||\, \mathbf{stop}_B \ ,$$

where

- $A_i = \{\, a \in \Sigma_i \mid \neg\exists r \in \mathcal{Y} : a = r[i] \,\}$ *for* $1 \leq i \leq n$;
- $\Phi_i = \{\, (a, r) \mid a \in \Sigma_i \wedge r \in \mathcal{Y} \wedge a = r[i] \,\}$ *for* $1 \leq i \leq n$;
- $A = \{\, r \in \mathcal{Y} \mid r[0] = \tau \,\}$;
- $\Phi = \{\, (r, a) \mid r \in \mathcal{Y} \wedge a = r[0] \neq \tau \,\}$; *and*
- $B = \{\, a \in \Sigma \mid \neg\exists r \in \mathcal{Y} : a = r[0] \,\}$.

Proof. The part using A_i removes those non-τ-transitions of M_i that have no matching rule. Φ_i renames the remaining non-τ-transitions of M_i so that $||$ synchronizes transitions that match the same rule. Then $\setminus A$ ensures that the final name is τ if the rule requires so. Φ fixes the final names that must not be τ, and $\mathbf{stop}_B$ adds to Σ the elements that are still missing. □

Together with the modelling of channels as state machines in the cash dispenser system, the results and examples in this subsection justify the following informal claims (however, please see also Sect. 5.3).

> All communication between state machines can be expressed in terms of ||, hiding, and relational renaming. Synchronization rules are a handy shorthand notation for that.

> Use of a local or shared variable is essentially parallel composition with a state machine that directly represents the variable.

Earlier versions of synchronization rules were presented in [1,17].

4.5 Input and Output

In the ?-, etc., notation, ? denotes input and ! output. If transitions labelled $\mathsf{a}!x$ and $\mathsf{a}?y$ synchronize, then it is appropriate to say that the former is an output and the latter input transition. The one who outputs determines the value of the event parameter, and the one who inputs is ready for just any value. The one who inputs usually stores the value in a variable, but this should not be seen as a fundamental property of input, because one may cheat by storing the value to a variable that is not used later.

The roles of input and output are not clear at the level of transitions, but are still at the level of individual event parameters, when transitions labelled $\mathsf{a}!1?x$ and $\mathsf{a}?y!3$ synchronize. The situation may be unclear even if there is only one event parameter, like with $[n\colon \geq 0]\mapsto \mathsf{a}?n$ and $[i\colon < 1]\mapsto \mathsf{a}?i$. Here the first transition determines that the value is at least 0 and the second that it is less than 1. So it is 0. However, neither transition determines the value alone, so neither can be called output. An even more confusing example is $[x := x]\mapsto \mathsf{a}?x$, because, although it seems to read the afterwards value of x from the event parameter, the guard forces the value to be the same as the original value. So it means precisely the same as $\mathsf{a}!x$.

In conclusion, process algebras allow forms of interaction where the notions of input and output do not make sense. Some interaction patterns may be very hard to implement in practice, especially in a distributed setting, but it is better to allow them in the theory than to rule out useful forms of interaction.

> Input and output are roles in interaction. Often interaction can be understood in terms of input and output, but not always.

4.6 Other Operators

Process-algebraic languages (such as [3,13,20]) have other operators in addition to variants of what we have already discussed. In this subsection we briefly discuss the most common. They are not necessary for most of the rest of this tutorial, but are central in many other writings on process algebras. In Sect. 5.3

they will be used to demonstrate that despite their great modelling power, synchronization rules do not cover all reasonable ways of building systems.

Action prefix $a;P$ (also written as $a.P$ and $a \to P$) means a system that first executes an a-transition and then behaves like P.

There are variants of the *choice* operator with somewhat different meanings. $P \,\square\, Q$, also written as $P + Q$, has initially the capability of behaving like P and like Q. Its first transition is either an initial transition of P or of Q, and then it continues like P or Q according to with whose transition it started. The environment does not directly see whether P or Q was chosen. Of course, if the action of the initial transition is visible and only used by Q, then it is possible to reason that Q was chosen. As an example of the variants of the choice operator, in *nondeterministic choice* $\sqcap$ the choice is done silently even if the initial transitions have different actions.

The *interrupt* operator $P \rhd Q$ is otherwise like choice, but Q has the ability to start until P has terminated successfully. Successful termination is indicated by executing a transition with a special action that has been reserved for this purpose. When Q has started, P cannot continue. A *divergence* is an infinite sequence of invisible events. It corresponds to a livelock. Of the operators that we have discussed, interrupt is the only one that can, roughly speaking, stop a divergence.

We used graphs to specify individual state machines, but many process-algebraic languages express everything in terms of textual expressions. Cyclic behaviour is specified by letting expressions call themselves recursively. In this setting, choice operators are the main means of specifying branching behaviour. For instance, BANK can be specified with the following expression.

$$\begin{array}{rcl}
\mathrm{BANK}(b) & = & \mathsf{Rq}?z;\,(\;\; [z > b]\!\!\mapsto \mathsf{Sno};\mathrm{BANK}(b) \\
& & \quad\square\;\; [z \le b]\!\!\mapsto \mathsf{Sy}!z;\,(\;\; \mathsf{uc};\mathrm{BANK}(b) \\
& & \qquad\qquad\square\;\; \mathsf{Rdone};\mathsf{r}!z;\mathrm{BANK}(b-z) \\
& & \qquad\qquad\;\;) \\
& & \quad)
\end{array}$$

This way of specifying systems makes it easy to model some situations that cannot be modelled easily or at all with our state machine formalism, like on-the-fly creation of new state machines. However, recursion complicates significantly the development of semantic theories like the ones in Sect. 5. As a consequence, some theories give unintuitive meanings to expressions like $P = \tau;P$. Furthermore, modellers of systems may find the notation cryptic and laborious to use. The present author believes that this is one of the reasons why process-algebraic methods have received much less attention than they deserve.

5 Abstract Behaviour

The detailed behaviour of a system with many hidden actions has typically few visible transitions and many τ-transitions. It cannot be drawn as a readable picture because of the many τ-transitions. In this section we discuss theories of

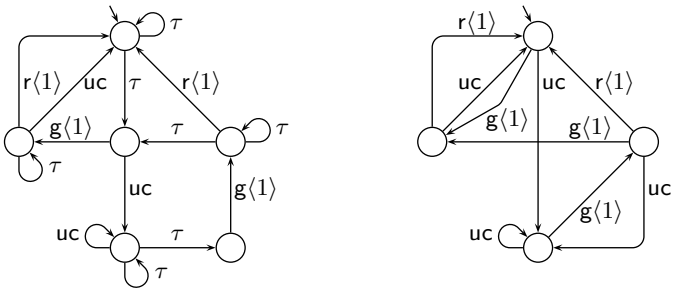


Fig. 8. The CFFD- and trace semantics version of the user's money view in Fig. 6 restricted to data type {1}

abstract behaviour, with which one can get rid of most τ-transitions and produce pictures such as Fig. 6. There are numerous such theories, so we concentrate on one and briefly mention some others. As a by-product we get a proof that the interrupt operator is fundamental instead of a shorthand. We also study the notion of determinism in the context of abstract behaviour. Finally we mention some verification techniques that exploit abstract behaviour.

5.1 Trace Semantics

A *trace* of an LTS is the sequence of visible actions that is obtained from any finite path that starts in the initial state. For instance, both LTSs in Fig. 8 have the traces ε, $\mathsf{g}\langle 1\rangle$, uc, $\mathsf{g}\langle 1\rangle\,\mathsf{r}\langle 1\rangle$, $\mathsf{g}\langle 1\rangle\,\mathsf{uc}$, $\mathsf{uc}\,\mathsf{uc}$, $\mathsf{uc}\,\mathsf{g}\langle 1\rangle$, $\mathsf{g}\langle 1\rangle\,\mathsf{r}\langle 1\rangle\,\mathsf{g}\langle 1\rangle$, and so on. ε denotes the empty sequence of visible actions. It is a trace of every LTS. A trace of a system is a trace of its behaviour. Figure 8 left has been made from Fig. 6 by restricting the type of x and y to $\{1\}$.

To make it easier to talk about traces and related things, let $s =\sigma\Rightarrow s'$ denote that there is a path from s to s' such that its sequence of visible actions is σ. By $s =\sigma\Rightarrow$ we mean the same thing but do not mention the end state of the path. With this notation, the set of traces of $L = (S, \Sigma, \Delta, \hat{s})$ is $Tr(L) = \{\,\sigma \mid \hat{s} =\sigma\Rightarrow\,\}$.

The *trace semantics* of L is the pair $(\Sigma(L), Tr(L))$, where $\Sigma(L)$ is the alphabet of L. Two systems are *trace equivalent* if and only if they have the same trace semantics, that is, the same alphabet and the same set of traces, that is, $\Sigma(L) = \Sigma(L') \wedge Tr(L) = Tr(L')$. Every semantics induces an equivalence — $L \simeq L'$ if and only if they have the same semantics. On the other hand, the equivalence classes of any equivalence can be thought of as a semantics. Therefore, we will sometimes use the words "semantics" and "equivalence" interchangeably.

The set of traces of a finite LTS is essentially the same thing as the language accepted by a finite automaton whose every state is a final state. As a consequence, well-known algorithms from automata theory can be used for manipulating finite LTSs so that the trace semantics is preserved. One may, for instance, construct the smallest deterministic LTS that has the same trace semantics as a given finite LTS. Figure 8 right shows the result of doing that to Fig. 8 left.

When we say that an equivalence $\simeq$ *preserves* a property we mean that for every L and L', $L \simeq L'$ implies that L and L' give the same value to the property. For instance, if $\simeq$ preserves deadlocks and $L \simeq L'$, then either none or both of L and L' may deadlock.

The major drawback of the trace equivalence is that it does not preserve deadlocks and livelocks. In those applications where this does not matter, trace semantics is excellent. What the trace equivalence preserves is called *stuttering-insensitive safety properties*. Safety properties are the properties whose violation can be detected after a finite execution, without knowing the future. Deadlock and livelock cannot be detected so, because if an external observer who only sees the visible events did not see anything happen, she does not know whether that was because she did not wait long enough or because nothing is going to ever happen. She cannot assume that if something is going to happen, it will happen in, say, 1000 time units, because there may be 1001 τ-events before the next visible event.

Stuttering-insensitive means that the number of τ-events before a visible event or deadlock does not matter. Bisimilarity is not stuttering-insensitive, but all semantics in this section are. Indeed, the usefulness of the abstract semantics comes from throwing away unnecessary information, and the number of τ-events is almost always unnecessary information.

Trace equivalence can preserve any stuttering-insensitive safety property, and does not preserve deadlock-freedom. Temporal logic researchers sometimes classify deadlock-freedom as a stuttering-insensitive safety property [19, p. 309]. We have a paradox!

Deadlock-freedom is expressed in temporal logics as $\Box(E_1 \vee \cdots \vee E_n)$, where $\Box$ means "always", and the E_i are equivalent to the enabling conditions of the atomic statements of the system. The formula can also be modelled in trace semantics. However, if one more statement is added to the system but not to the formula, the formula is still meaningful, but does not any more express deadlock-freedom. We see that classification of deadlock-freedom as a safety property assumes that the program code of the system is available, to know which formula expresses deadlock-freedom. Process algebra researchers do not make that assumption, so they can model the formula but cannot know if it expresses deadlock-freedom.

State propositions can be taken into account in traces by adding $val(\hat{s})$ to the semantics and replacing the actions by pairs $\langle a, P\rangle$, where $a \in \Sigma \cup \{\tau\}$ and $P \subseteq \Pi$ [11]. P lists the propositions whose truth values change during the transition. The pair $\langle \tau, \emptyset\rangle$ plays the role of the invisible action. In this formalism, a trace is a sequence of pairs $\langle a, P\rangle$, where $a \neq \tau$ or $P \neq \emptyset$.

5.2 Stable Failures

A deadlock-preserving equivalence is obtained by extending the trace semantics with the set of *stable failures*. We say that a state *refuses* a set A of actions, if none of its output transitions is labelled with an element from A. Thus a deadlock is a state that refuses $\Sigma \cup \{\tau\}$, where Σ is the alphabet of the system.

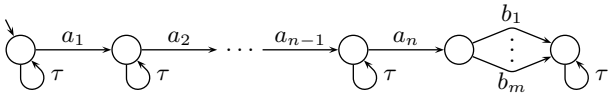


Fig. 9. Showing that stable failures are necessary to preserve deadlocks

A stable failure is a pair (σ, A), where σ is a trace and $A \subseteq \Sigma$. The system has (σ, A) as its stable failure if and only if it has an execution whose trace is σ and that ends in a state that refuses $A \cup \{\tau\}$. The set of stable failures of L is thus

$$SFail(L) = \{\ (\sigma, A) \mid \exists s : \hat{s} =\sigma\Rightarrow s \wedge A \subseteq \Sigma(L) \wedge \forall a \in A \cup \{\tau\} : \neg(s -a\rightarrow)\ \}\ .$$

Here $s -a\rightarrow$ means that there is an s' such that $s -a\rightarrow s'$.

The motivation for this complicated-looking notion is that its presence in the semantics either explicitly or implicitly is necessary to preserve deadlocks, if we use $||$. Let $\sigma = a_1 a_2 \cdots a_n \in \Sigma(L)^*$ and $A = \{b_1, \ldots, b_m\} \subseteq \Sigma(L)$. Consider the LTS L_A^σ in Fig. 9 with the alphabet $\Sigma(L)$. If $L \,||\, L_A^\sigma$ has executed some other trace than σ, then τ is enabled. If σ has been executed and L can execute τ or any of $b_1, \ldots, b_m$, then that action is enabled. Otherwise nothing is enabled. So $L \,||\, L_A^\sigma$ deadlocks if and only if both execute σ and then L refuses τ and $b_1, \ldots, b_m$. That is possible if and only if (σ, A) is a stable failure of L.

We mentioned in Sect. 4.1 that an equivalence $\simeq$ is a congruence with respect to an operator f for putting state machines or behaviours together, if and only if for every $L_1, \ldots, L_n, L'_1, \ldots, L'_n$ we have that $L_1 \simeq L'_1$, $\ldots$, $L_n \simeq L'_n$ imply $f(L_1, \ldots, L_n) \simeq f(L'_1, \ldots, L'_n)$. Let $L_1 \simeq L_2$, where $\simeq$ preserves deadlocks and the alphabet, and is a congruence with respect to $||$. Because it is a congruence, $L_1 \,||\, L_A^\sigma \simeq L_2 \,||\, L_A^\sigma$. Then $(\sigma, A) \in SFail(L_1)$ if and only if $L_1 \,||\, L_A^\sigma$ has a deadlock if and only if $L_2 \,||\, L_A^\sigma$ has a deadlock if and only if $(\sigma, A) \in SFail(L_2)$. We have proven the following.

Proposition 18. *Any congruence with respect to $||$ that preserves the alphabet and deadlocks also preserves stable failures.*

This, actually simple, result is from [28]. The semantics consisting of the alphabet and stable failures (but not traces) is a congruence with respect to $||$, hiding, relational renaming, and action prefix. If also the choice operator $\square$ is used, then the so-called "initial stability" bit that we will discuss a bit later must be added to the semantics, to retain the congruence property. If, furthermore, the interrupt operator $\triangleright$ is used, then also traces must be added to the semantics. These emphasize that the congruence property is sensitive to the set of operators in use.

The discussion above can be summarized by saying that even if we could directly observe only deadlocks, we could get information on stable failures and perhaps also other things by putting the system to a suitable environment, and observing the deadlocks of the result. The semantic model consisting of the alphabet, traces, and deadlocks is not *fully abstract*, because we could get additional information about the system by using it as a component in a bigger

Fig. 10. Illustrating a congruence problem with failures

system. On the other hand, assuming that only ||, hiding, relational renaming, and action prefix are allowed for connecting the system to its environment, the semantic model consisting of the alphabet and stable failures is fully abstract: it contains precisely the information that we can get by putting the system under test in a suitable environment and then observing deadlocks.

Many advanced process-algebraic verification methods are based on replacing components of a system by equivalent components that are better suited for continuing the verification. For instance, an LTS may be replaced by a smaller but equivalent LTS. The correctness of this relies on the assumption that the equivalence in use is a congruence, implying that the semantics of the system as a whole does not change in the replacement. Therefore, the congruence property is central.

When $\neg(s -\tau\!\!\rightarrow)$ and $a \in \Sigma$, then $s =\!a\!\Rightarrow$ and $s -\!a\!\!\rightarrow$ mean the same. As a consequence, stable failures can be defined equivalently as

$$\{\ (\sigma, A) \mid \exists s : \hat{s} =\!\sigma\!\Rightarrow s \wedge A \subseteq \Sigma \wedge \forall a \in A : \neg(s =\!a\!\Rightarrow) \wedge \neg(s -\tau\!\!\rightarrow)\ \}\ .$$

Historically there was great interest in *failures*, defined as

$$\{\ (\sigma, A) \mid \exists s : \hat{s} =\!\sigma\!\Rightarrow s \wedge A \subseteq \Sigma \wedge \forall a \in A : \neg(s =\!a\!\Rightarrow)\ \}\ .$$

This line of research ran into trouble because of the congruence problem that is illustrated in Fig. 10. The LTSs in the figure have the same alphabet, traces, and failures. However, if a is hidden in both, then $(\varepsilon, \{b\})$ becomes a failure of the one on the right but not of the other. The semantics consisting of the alphabet, traces, and failures is thus not a congruence with respect to hiding.

This is why failures had to be replaced by stable failures. A state is called *stable* if and only if it cannot execute τ. The difference of failures and stable failures is that in the latter, the state after the trace must be stable.

The desire that the equivalence must be a congruence has led to numerous small variants of equivalences. For instance, the semantics consisting of the alphabet, traces, and stable failures stops from being a congruence when the choice operator is employed. This problem can be solved simply by adding one bit to the semantics, known as the *initial stability bit*. It tells whether the initial state is stable, that is, whether $\neg(\hat{s} -\tau\!\!\rightarrow)$.

> The congruence property is sensitive to the set of operators in use. This is one reason why there are so many semantic models in process algebras.

Information on stable failures can be taken into account in algorithms by attaching to each relevant state a set of minimal *acceptance sets*. An acceptance

set is the complement, with respect to the alphabet, of a set that the state refuses. Acceptance sets carry the same information as refused sets but tend to be smaller. All stable failures represented by a non-minimal acceptance set are also represented by their smaller acceptance sets, so storing non-minimal acceptance sets would be pointless.

5.3 On Building Operators from Other Operators

Often in computer science, an operator can be thought of as just an abbreviation of an expression written without using it, while another operator genuinely adds to the expressivity of the language. For instance, if the propositional operators $\wedge$ and $\neg$ are available, then $\vee$ is obtained as $\varphi \vee \eta \Leftrightarrow \neg(\neg\varphi \wedge \neg\eta)$, but if only $\wedge$ and $\vee$ are available, then $\neg$ cannot be constructed. In Sect. 4 we demonstrated that $||$, hiding, and relational renaming suffice to represent both all forms of communication and the use of local or shared variables. However, this does not imply that all reasonable operators or all reasonable ways of building systems could be built from them. This subsection is devoted to this issue.

First we have to discuss what do we mean by representing an operator as a function of other operators. We use the interrupt operator $\rhd$ as an example. If f is built from other operators than $\rhd$, and if $f(L_1, L_2)$ is isomorphic to $L_1 \rhd L_2$ for every LTSs L_1 and L_2, then it is clear that $L_1 \rhd L_2$ can be built from the other operators. Intuition might suggest that it is not possible, and we will soon see that it is indeed the case.

However, requiring isomorphism is usually unnecessarily strict. For instance, if we are only interested in the trace semantics, then it suffices that $f(L_1, L_2)$ has the same trace semantics as $L_1 \rhd L_2$. Indeed, such an f can be constructed only using $||$ and relational renaming. For simplicity, we ignore the issue of successful termination, although taking it into account would not be difficult.

Let Σ_1 and Σ_2 be the alphabets of L_1 and L_2, respectively. For $1 \leq i \leq 2$, let Φ_i rename each $a \in \Sigma_i$ to (a, i). The alphabets of $L_1\Phi_1$ and $L_2\Phi_2$ are disjoint. Let Σ be their union. Let Φ rename each $(a, 1)$ and each $(a, 2)$ in Σ to a. Let L be the LTS who has two states $\hat{s}_L$ and s_L, whose alphabet is Σ, and whose transitions are $\{\ (\hat{s}_L, (a, 1), \hat{s}_L)\ \mid\ a \in \Sigma_1\ \} \cup \{\ (\hat{s}_L, (a, 2), s_L)\ \mid\ a \in \Sigma_2\ \} \cup \{\, (s_L, (a, 2), s_L) \mid a \in \Sigma_2 \,\}$. Each visible transition of $(L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi$ consists of a visible transition of L and either L_1 or L_2 (but not both). Clearly L always allows L_2 to execute visible transitions. On the other hand, L allows L_1 to execute visible transitions only as long as L_2 has not executed any.

We see that $(L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi$ behaves otherwise like $L_1 \rhd L_2$, except that it is not the starting of L_2 but the first visible transition of L_2 that stops L_1 from executing visible transitions, and nothing stops L_1 from executing invisible transitions. However, these differences do not affect the traces. Therefore, $\Sigma((L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi) = \Sigma(L_1 \rhd L_2)$ and $Tr((L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi) = Tr(L_1 \rhd L_2)$.

On the other hand, we will now show that if the equivalence preserves the alphabet and stable failures, then there is no function f such that $f(L_1, L_2)$ is equivalent to $L_1 \rhd L_2$ and f can be built from only $||$, hiding, relational renaming, and action prefix. Such equivalences include isomorphism and bisimilarity. They

also include the CFFD-equivalence and the divergence-preserving variants of weak and branching bisimilarity mentioned later in this section.

Proposition 19. *Let $\simeq$ be an equivalence that preserves the alphabet and stable failures. There is no function f that can be built from only parallel composition, hiding, relational renaming, and action prefix, such that $L_1 \rhd L_2 \simeq f(L_1, L_2)$ for every LTSs L_1 and L_2. A similar result holds if also the choice operator may be used in building f and $\simeq$ also preserves initial stability.*

Proof. Assume that f exists. Let $\approx$ be defined by $L \approx L'$ if and only if $\Sigma(L) = \Sigma(L')$ and $SFail(L) = SFail(L')$. With the choice operator, $\approx$ also requires that the initial state of either none or both of L and L' is stable. By [28], $\approx$ is a congruence with respect to the mentioned operators. Therefore, if $L_1 \approx L_1'$ and $L_2 \approx L_2'$, then $L_1 \rhd L_2 \simeq f(L_1, L_2) \approx f(L_1', L_2') \simeq L_1' \rhd L_2'$, yielding $L_1 \rhd L_2 \approx L_1' \rhd L_2'$. This means that $\approx$ is a congruence with respect to $\rhd$, which is in contradiction with [28]. □

Proposition 18 lets us to state the above result as follows: the interrupt operator cannot be built from other common process-algebraic operators, if the semantics in use preserves the alphabet and deadlocks and is a congruence with respect to $||$. However, we saw that with the trace semantics it was possible. We conclude that whether or not an operator can be built from other operators depends on the semantics in use. We also conclude the following:

> Despite the great expressive power of $||$, hiding, and relational renaming, they cannot model all useful ways of building systems.

5.4 CFFD-Semantics

Information on livelocks can be added to the semantics in the form of *divergence traces*. We say that state s *diverges* if and only if an infinite sequence of τ-events can be started at s. A divergence trace is a trace that ends in a diverging state. For instance, all traces of Fig. 8 left are divergence traces. The set of divergence traces is denoted with $DivTr(L)$.

When divergence traces are added, then also something else must be done to the semantics to maintain the congruence property. One possibility is to add the *infinite traces* $InfTr(L)$. They are the infinite sequences of visible actions that arise from infinite executions that start at the initial state. The resulting semantics is called *Chaos-free failures divergences semantics* or *CFFD-semantics* [40]. We will explain the odd name soon.

The definition of CFFD-semantics as usually presented in the literature lacks the trace component. This is because it can be derived from other components: $Tr(L) = DivTr(L) \cup \{\, \sigma \mid (\sigma, \emptyset) \in SFail(L) \,\}$. So it is implicitly present even if it is not explicitly mentioned. If the LTSs are finite, also infinite traces can be left out for a similar reason. On the other hand, the initial stability bit must be added if the choice or interrupt operator is used. This is why variants of the CFFD-semantics have been presented in the literature.

An alternative way of solving the congruence problem caused by divergence traces is to only consider *minimal* divergence traces and ignore everything after them. A divergence trace is minimal if and only if none of its proper prefixes is a divergence trace. This is called *catastrophic divergence*, and a process that diverges initially is sometimes called *chaos*.

This solution arises naturally from the mathematics used for giving a meaning to recursive process definitions of the kind in Sect. 4.6. It was chosen as the main semantics of the well-known theory of *Communicating Sequential Processes* (*CSP*) [13,23,25]. Unfortunately, it implies that, for instance, the LTS in Fig. 8 left is equivalent to chaos. Thus no information on its behaviour other than that it diverges initially is preserved. This is a big drawback in many applications. Therefore, when CFFD-semantics was invented, its name was chosen to emphasize the similarity to CSP and the absence of chaos. Also CSP researchers admit that it would be nice to see beyond divergence [24].

Figure 8 left has been produced using CFFD-semantics. So the information that it gives on the deadlocks and livelocks of the system is real. There are no deadlocks, but sometimes the system can be in a state where it only can execute $\mathsf{g}\langle 1\rangle$ or uc, and sometimes only $\mathsf{g}\langle 1\rangle$ is possible. For instance, we can reason from the figure that if the user refuses to take the money, BANK will eventually execute uc if it has not done that already. This is because according to Fig. 2, ATM cannot send the "done" message before the user has taken the money. The many livelocks in Fig. 8 arise from the possibility of the user trying again and again, repeatedly getting loss of connection. If a yes-answer gets through to ATM, then ATM must execute the $\mathsf{g}\langle 1\rangle$-transition to continue, which is seen in Fig. 8 as commitment to $\mathsf{g}\langle 1\rangle$ perhaps together with uc.

Readers of the figures produced with CFFD-semantics — the present author included — are sometimes confused by questions like the following. Absence of livelock implies that ATM is in its "yes"-branch. From there, all paths to livelocks go via the $\mathsf{g}\langle 1\rangle$-transition. There is no livelock in the centre state of Fig. 8 but there is in the bottom middle state, and the transition linking these two states is labelled by uc and not $\mathsf{g}\langle 1\rangle$. What went wrong in the reasoning?

The answer to this question is that CFFD-semantics does not preserve information about deadlocks and livelocks in other states of an execution than the last. Therefore, deadlocks and livelocks must only be analysed at the end states of executions, not during intermediate states. If the execution ends at the bottom middle state, then one must read deadlocks and livelocks only there, not in the centre state of the figure.

It would be possible to draw an LTS from which deadlock and livelock information could be read also in the middle of an execution. However, it would be bigger than the one in Fig. 8 left. There is a trade-off.

The more information is preserved, the bigger are the resulting LTSs.

This principle works both when choosing the set of visible actions and when choosing the semantics.

A variant of CFFD-semantics called *NDFD-* or *nondivergent failures divergences* semantics [14,33] is the weakest congruence that preserves all stuttering-insensitive properties expressible in classical linear temporal logic [19]. It does not preserve deadlocks, except when the congruence requirement forces it to do so. CFFD-semantics preserves the same and also deadlocks. This makes it useful for many but not all applications.

CFFD- and NDFD-semantics suffer from the same problem as process-algebraic methods in general: it is difficult to express so-called *fairness assumptions* that are used in temporal logics to guarantee progress. With fairness assumptions one could, for instance, remove livelocks in Fig. 8 left. Promising ideas towards solving this drawback were presented in [21,22], but, unfortunately, nobody has continued that research.

5.5 CFFD-Preorder

A *preorder* is a reflexive and transitive binary relation. A *precongruence* with respect to f is a preorder $\preceq$ such that if $L_1 \preceq L'_1, \ldots, L_n \preceq L'_n$, then $f(L_1, \ldots, L_n) \preceq f(L'_1, \ldots, L'_n)$. Every preorder induces an equivalence and every precongruence induces a congruence by $L \simeq L' \Leftrightarrow L \preceq L' \wedge L' \preceq L$. A precongruence that induces CFFD-equivalence is obtained by $L \preceq L' \Leftrightarrow \Sigma(L) = \Sigma(L') \wedge \mathit{SFail}(L) \subseteq \mathit{SFail}(L') \wedge \mathit{DivTr}(L) \subseteq \mathit{DivTr}(L') \wedge \mathit{InfTr}(L) \subseteq \mathit{InfTr}(L')$. We call it *CFFD-preorder*. The reason for requiring equality of alphabets instead of the subset relation is too technical to be discussed here [33].

Let $\preceq$ denote CFFD-preorder. If $L \preceq L'$, then whatever trace, stable failure, divergence trace, or infinite trace L can do, also L' can do, but not necessarily vice versa. In particular, if L' cannot do anything wrong — cannot execute a wrong visible action, cannot deadlock when it is not allowed to, and cannot livelock when it is not allowed to — then also L cannot do anything wrong. So correctness of L' implies the correctness of every L that satisfies $L \preceq L'$. Indeed, there is a theorem saying that if L' satisfies a stuttering-insensitive linear temporal logic formula and $L \preceq L'$, then also L satisfies the formula [33].

This implies that we need not know the components of a system precisely when verifying the correctness of the system. Instead, if we have many possible alternatives for a component, it suffices that we use those among them that are the biggest in CFFD-preorder. This is particularly important when also the users of the system are modelled. Sometimes the correctness of a system depends on the users to obey some rules. It is often easy to model the CFFD-biggest user that obeys the rules. If the system works correctly with it, then it works correctly with all users that obey the rules.

This also means that often verification does not consist of checking whether the system is equivalent to the specification but whether the system is at most the specification. Systems are often allowed to be better than their specifications. If we buy a fifo queue with capacity 3 and get a fifo queue with capacity 4 for the same price, we do not mind, although it is not equivalent to the specification.

Checking CFFD-equivalence of two LTSs is **PSPACE**-complete. Checking CFFD-preorder is **PSPACE**-complete in the size of the specification LTS but

polynomial time in the size of the system LTS. This is fortunate, because the system LTS is usually a parallel composition of components and thus much bigger than the specification LTS. Similar remarks apply to trace semantics, CSP, and linear temporal logic. In Sect. 5.6 we will see that all these semantics are called "linear time".

> Verification of linear-time properties is typically polynomial time in the size of the state space of the system and **PSPACE**-complete in the size of the state space of the specification.

To get a feeling of CFFD-preorder, let us discuss its extreme elements. For every alphabet Σ, there is a maximum element, that is, L' such that every L with the same alphabet satisfies $L \preceq L'$. It has an initial state s_1 and another state s_2, and the transitions $s_1 -\tau\rightarrow s_2$, $s_1 -\tau\rightarrow s_1$, and $s_1 -a\rightarrow s_1$ for every $a \in \Sigma$. Its traces are Σ^*, all traces are divergence traces, and (σ, Σ) is a stable failure for every trace σ. Also its set of infinite traces is maximal.

On the other hand, there is no minimum element. The LTS that has no transitions has no divergence traces, while the LTS that has a τ-transition from its initial state to itself but no other transitions has no stable failures. Thus a minimum element must have no divergence traces and no stable failures. However, ε is a trace of every LTS, so each LTS has the divergence trace ε or the stable failure $(\varepsilon, \emptyset)$. According to the above-mentioned theorem about CFFD-preorder and linear temporal logic, a minimum element would satisfy all satisfiable stuttering-insensitive formulas. It would thus satisfy all satisfiable specifications. It would be a single system that is good for everything! It is a sign of the healthiness of our theory that such a system does not exist.

With trace preorder, the LTS that has no transitions is a minimum element. Indeed, it satisfies all specifications that can be formulated in trace semantics. Trace semantics only preserves stuttering-insensitive safety properties. Safety properties require that the system must never do anything wrong. The LTS that has no transitions does not ever do anything wrong, because it does not ever do anything. We see that a specification formalism is not complete unless it can specify that the system must do something. Trace preorder cannot do that, but CFFD-preorder can, by disallowing divergence traces and stable failures appropriately.

Although CFFD-preorder has no minimum element, it has minimal elements. It is useful to know that if σ is a trace, then it is a divergence trace or $(\sigma, \emptyset)$ is a stable failure or both. If $(\sigma, \emptyset)$ is a stable failure, then, for every visible action a, σa is a trace or $(\sigma, \{a\})$ is a stable failure or both. Minimal elements are obtained by avoiding the option "both" and by restricting divergence traces to the minimal ones. We skip the proof (infinite traces cause some trouble).

Proposition 20. *An LTS L is CFFD-minimal if and only if for every $\sigma \in Tr(L)$ either*

- $(\sigma, \emptyset) \notin SFail(L)$ *and for every* $a \in \Sigma(L)$, $\sigma a \notin Tr(L)$; *or*
- $\sigma \notin DivTr(L)$ *and for every* $a \in \Sigma(L)$, $\sigma a \notin Tr(L)$ *or* $(\sigma, \{a\}) \notin SFail(L)$.

That is, if L livelocks immediately after some trace σ, then it cannot do anything else nor refuse anything after σ; and if it does not livelock immediately after σ, then, for each visible action a, it can execute a as the next visible action after either no or every way of executing σ. An important theme here is that if L can do or refuse something after one way of executing a trace, then it can do or refuse the same after every way of executing the same trace. In other words, *CFFD-minimal systems are deterministic.*

This statement is not a theorem but an intuitive statement, because we have not made it precise what deterministic means, but rely on intuition. The classical definition used in automata theory does not apply, because, for instance, it declares nondeterministic the LTS $s_1 \leftarrow a - \hat{s} - a \rightarrow s_2$. We will return to this issue in Sect. 5.7. There we can also tackle the opposite question, that is, are all deterministic systems CFFD-minimal.

Roughly speaking, the smaller a system is in CFFD-preorder, the more deterministic it is, and vice versa.

We did not model the user in the cash dispenser system. The unmodelled user corresponds to the LTS that has one state, a transition for each $a \in \Sigma$ from that state to itself, and nothing else. Proposition 20 implies that this user is not the most general reasonable user. It is CFFD-minimal and thus only represents itself in verification. The most general reasonable user of the cash dispenser system can be modelled as an LTS with three states and the transitions $\hat{s} - \mathsf{ci} \rightarrow s_1$, $\hat{s} - a \rightarrow \hat{s}$ for $a \in \Sigma \setminus \{\mathsf{ci}\}$, $\hat{s} - \tau \rightarrow s_2$, $s_1 - \mathsf{co} \rightarrow \hat{s}$, and $s_1 - a \rightarrow s_1$ for $a \in \Sigma \setminus \{\mathsf{co}\}$, where Σ consists of ci, co, nm, lc, and $\mathsf{w}\langle i \rangle$ and $\mathsf{g}\langle i \rangle$ for every $i \in \mathbb{N}$. This differs from the unmodelled user in that it can deadlock when the card is not in, modelling the possibility of the user going away and never again trying to withdraw money. The model involves the assumption that the user will not go away while the card is in.

For reasoning about the progress properties of the system from the point of view of the user, it is a good idea to make ci and co visible and the remaining actions invisible. With the unmodelled user, this produces the two-state LTS where ci and co alternate. With the model of the user described above, a deadlock is added to the initial state of the previous result. From these it is clear that the system does not livelock, and it deadlocks only when the user goes away while the card is not in.

The modelling of components and the interpretation of the resulting pictures when using CFFD-semantics was discussed in detail in [38].

A specification of a system or an assumption about its component must often be nondeterministic, to leave enough room for different valid implementations and users.

Often preorders are more appropriate than equivalences for comparing systems against specifications.

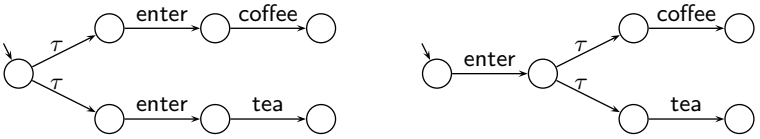


Fig. 11. Two CFFD-equivalent but not observation equivalent LTSs

5.6 Weak and Branching Bisimilarity

A famous abstract equivalence that is not based on sets of traces or failures is the *observation equivalence* of CCS, also known as *weak bisimilarity* [20]. Its definition resembles the definition of bisimilarity, but uses the $=\cdots\Rightarrow$-relation. Every visible action, possibly preceded and followed by invisible actions, must be simulated by a visible action, possibly preceded and followed by invisible actions. Also every (possibly empty) sequence of invisible actions must be simulated by a (possibly empty) sequence of invisible actions.

Definition 21. *Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS. The relation "$\simeq$" $\subseteq S \times S$ is a* weak bisimulation*, if and only if for every $s_1 \in S$, $s_2 \in S$, $s \in S$, and $a \in \Sigma \cup \{\varepsilon\}$ such that $s_1 \simeq s_2$ the following hold:*

- *If $s_1 =a\Rightarrow s$, then there is $s' \in S$ such that $s_2 =a\Rightarrow s'$ and $s \simeq s'$.*
- *If $s_2 =a\Rightarrow s$, then there is $s' \in S$ such that $s_1 =a\Rightarrow s'$ and $s' \simeq s$.*

Two LTSs are observation equivalent *if and only if they have the same alphabet and their disjoint union has a weak bisimulation such that the initial states simulate each other.*

The choice and interrupt operators cause a congruence problem also to observation equivalence, so a variant known as *observation congruence* has been defined. Another variant is obtained by making the equivalence sensitive to livelocks, by requiring that if $s_1 \simeq s_2$, then either neither or both of s_1 and s_2 diverge. This equivalence is strictly stronger than CFFD-equivalence.

Observation equivalence is a *branching time* concept, while CFFD-equivalence is *linear time*. That is, individual executions and properties of their end states suffice for checking CFFD-equivalence, while observation equivalence requires a tree-like structure (or graph). Figure 11 left shows a professor who silently chooses between coffee and tea, and then enters a cafeteria and takes what she chose. The one on the right is otherwise similar, but makes the choice after entering the cafeteria (but without ensuring that both are available). They are CFFD-equivalent but not observation equivalent.

In Definition 21, the first and last state of the simulating execution must simulate the first and last state of the simulated execution, but the intermediate states need not simulate any states. In *branching bisimilarity* [41], also the intermediate states must simulate states along the simulated sequence. It is strictly stronger than observation equivalence. In its extension that takes divergences into account, it does not suffice that diverging states are simulated

by diverging states. Instead, each infinite sequence of invisible actions must be simulated by an infinite sequence of invisible actions. The resulting equivalence preserves [5] stuttering-insensitive computation tree logic [6] similarly to how CFFD-equivalence preserves classical stuttering-insensitive linear temporal logic.

5.7 Operational Determinism

"Deterministic" has an established definition in automata theory. A direct translation of the definition to LTSs is that an LTS $(S, \Sigma, \Delta, \hat{s})$ is deterministic if and only if it has no τ-transitions, and for every $a \in \Sigma$, $s \in S$, $s_1 \in S$, and $s_2 \in S$, if $s -a\rightarrow s_1$ and $s -a\rightarrow s_2$, then $s_1 = s_2$. This definition is not useful in process algebras, because it deems very few LTSs deterministic, and determinism is not preserved by bisimilarity. For instance, the LTSs $\hat{s} -a\rightarrow s_1$ and $s_1 \leftarrow a- \hat{s} -a\rightarrow s_2$ are bisimilar but only the former is deterministic.

Motivated by the discussion in Sect. 5.5, we could define that an LTS is deterministic if and only if it is CFFD-minimal. This is similar to the definition in CSP [23], except that there divergence is treated differently. However, we would like the definition not be tied to any particular semantics. Furthermore, we would like the definition to deem deterministic as many LTSs as possible, because we will soon present a proposition whose usefulness benefits from that. The following notion [10] is suitable.

Definition 22. *LTS $(S, \Sigma, \Delta, \hat{s})$ is* operationally deterministic*, if and only if for every $\sigma \in \Sigma^*$, $s_1 \in S$, and $s_2 \in S$, if $\hat{s} =\sigma\Rightarrow s_1$ and $\hat{s} =\sigma\Rightarrow s_2$, then the following hold:*

- *for every $a \in \Sigma$, if $s_1 =a\Rightarrow$, then $s_2 =a\Rightarrow$; and*
- *if s_1 diverges, then s_2 diverges.*

CFFD-minimal LTSs are precisely the LTSs that are operationally deterministic and satisfy the following condition: for every $\sigma \in DivTr(L)$ and $a \in \Sigma$, $\sigma a \notin Tr(L)$. That is, Proposition 20 requires that after executing a divergence trace, the LTS cannot do anything else than diverge; but Definition 22 does not require so.

Now we can state a proposition about operationally deterministic LTSs. Again, we skip the proof [10].

Proposition 23. *If LTSs L and L' are operationally deterministic, $\Sigma(L) = \Sigma(L')$, $Tr(L) = Tr(L')$, and $DivTr(L) = DivTr(L')$, then L and L' are branching bisimilar, divergence-preserving branching bisimilar, observation equivalent, divergence-preserving observation equivalent, CFFD-equivalent, NDFD-equivalent, and CSP-equivalent.*

This result has the practical application that if an LTS is operationally deterministic (and that can be tested very efficiently), then it can be processed with any algorithm that preserves the alphabet, traces, divergence traces, and operational

determinism, and the result is valid in each of the mentioned semantics. It has also the philosophical message that a multitude of semantics in process algebras collapses if systems are operationally deterministic. A similar result for semantics that ignore divergences, that does not assume that $DivTr(L) = DivTr(L')$, was developed in [7,42]. Unfortunately, to make the collapse also cover congruences with respect to the choice operator, something extra is needed. For instance, the requirement that $\hat{s}$ is stable may be added to the formulation of operational determinism.

That there are so many different semantics in process algebras is largely because operational nondeterminism is an important feature in concurrency. For instance, if all systems were operationally deterministic, the distinction between linear time and branching time would disappear.

5.8 Verification Techniques

In this subsection we mention some verification techniques that are related to the theory in this section. Detailed information can be found in the cited sources, and in many cases also in the tutorials [30,32,34].

A basic method is *compositional LTS construction*. It means putting some components of the system together, reducing their joint behaviour, putting the result together with the reduced behaviour of a neighbouring subsystem and so on, until a reduced behaviour of the system as a whole is obtained. The semantics that is used must be a congruence with respect to the operators used in building the system. The first explicit mentionings of this idea are perhaps in [27,18], but the idea is so obviously built into process-algebraic theories that it has certainly been known before that.

Reducing the behaviour means applying some algorithm to the LTS that produces an equivalent but (hopefully) smaller LTS. With bisimilarity-based equivalences, there is a unique smallest equivalent LTS, and it can be found in polynomial time [15]. With trace- and failure-based equivalences, smallest equivalent LTSs are not necessarily unique, and finding one is **PSPACE**-hard. The problem is a generalization of the problem of finding a minimal (not necessarily deterministic) finite automaton that accepts the same language as a given finite automaton.

Fortunately, it is not necessary to find a minimal LTS, it suffices that it is equivalent and small. So one can use heuristic algorithms that run in polynomial time. Furthermore, algorithms based on the well-known determinization and minimization algorithms of finite automata have been extended to the failure semantics world, and they have been reported to run reasonably well in practice, e.g., [4,23,39]. Please see [29] for more comments on the relative efficiency of verification using bisimulation-based vs. failure-based semantics.

Stubborn set methods save effort during the computation of parallel composition by leaving out orderings of events that have the same effect as other orderings that are not left out. This type of methods are also called "partial order". Stubborn set methods for the major process-algebraic semantics were presented in [31].

Independently of the semantics, compositional LTS construction suffers from the *spurious behaviour* problem. That is, the behaviour of a subsystem may be much bigger than the behaviour of the system as a whole. This is because systems often obey invariants that strongly restrict the possible combinations of local states of the components, but, in isolation, subsystems do not necessarily obey them. Consider a fifo queue of capacity k that can store two different messages. In isolation it has $1 + 2 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$ states. When the same queue is in the well-known alternating bit protocol [2], there can be at most one place where successive messages are different. This reduces the number of possible states to k^2+k+1 (one empty queue, $2k$ with only one type of messages, and $2 \cdot \frac{1}{2}k(k-1)$ with two message types).

The spurious behaviour problem was pointed out and a solution for observation equivalence was presented in [9]. A general solution that applies to many semantics was presented in [16]. A key idea in these solutions is an *interface specification*, also known as *interface process*. It represents an assumption about the behaviour of the subsystem when it is within the system as a whole. For instance, one may represent the assumption that there is at most one place in the queue where successive messages are different. If the assumption is incorrect, then that is detected at the end of compositional LTS construction. Independently of whether it is correct, the interface process reduces the LTS of the subsystem. Interface processes require the addition of the notion of "undefined" to the semantics, but [16] shows how it can be done with very little need to rewrite tools.

If the result of compositional LTS construction is small, it can be analysed visually, like we did for Fig. 6 and 8. If it is small or big, one can compare it to a reference LTS with an equivalence or preorder checking algorithm. Preorder checking has the advantage that it can be done *on-the-fly*, that is, simultaneously with the computation of the behaviour of the system. This is a big advantage, because incorrect systems tend to have lots of spurious behaviour that the corresponding correct systems do not have. With on-the-fly verification, the computation may be terminated when the first counter-example has been found, so that most of the spurious behaviour will not be computed. Preorder checking is the major verification method with the *FDR* (Failures-Divergences Refinement) tool [23]. A CFFD-preorder version of this idea was presented in [12].

Some systems contain many identical components. Sometimes the behaviour of a subsystem with $n + 1$ components turns out equivalent to the behaviour with n components. (The probability of this happening can be increased with interface processes.) Then a simple induction argument yields that the behaviour of the system is the same for all numbers of the replicated components starting from n. Particularly intriguing applications of this idea were presented in [37], such as proving that the behaviour of a protocol is independent of the (finite) maximum number of times that it may re-transmit a message before giving up. The idea can also be used with precongruences: if $L \preceq L'$ and $f(L', K) \preceq L'$, then $f(L, K) \preceq f(L', K) \preceq L'$, yielding $f(f(L, K), K) \preceq f(L', K) \preceq L'$ and $f(\cdots f(f(f(L, K), K), K) \cdots, K) \preceq L'$ for any number of K-components. Outside process algebras, the idea has been presented in [43], among others.

In [26] it was pointed out that the notion of determinism is related to computer security. Consider a system with a trusted user and an untrusted user. The untrusted user must get no information about the behaviour of the trusted user. This holds, if the untrusted user's view to the system is deterministic. We already pointed out that there is a fast algorithm for checking determinism.

6 Conclusions

Compositionality at the structural level is routine in computer science and software engineering. There are modules, classes, and other kinds of units, and they can be nested. There are hierarchical Petri nets. The situation with compositionality of concurrent systems at the semantic level is confusing. On one hand, the idea is natural and it seems that it attracts many researchers. On the other hand, many widely valid basic facts have been found by process algebra research well before the year 2000, but seem little known.

The present author believes that one, but not the only, reason why process-algebraic results have failed to break through is that process-algebraic languages are cryptic and lead to cryptic fixed-point theories of semantics.

In this tutorial we have tried to make it clear that the semantic models are not tied to the cryptic languages, but apply to concurrent systems in general. We replaced recursion-based definitions of individual processes by state machines. For composing the system from its components, a small and natural set of operators was employed, and it was shown that in the end there are just synchronization patterns where some components do not participate, each one who participates does that by executing a modeller-chosen visible action, and the result has a modeller-chosen (not necessarily visible) action. Synchronous communication may seem unnatural and restricted, but we pointed out that it is the raw material from which all kinds of communication could be constructed. On the other hand, our formalism does not cover all reasonable ways of building systems, such as on-the-fly creation and abortion of processes.

The absence of event parameters from the definition of LTSs does not make the semantic theory incapable of processing them. It is just that the semantic theory is insensitive to event parameters, so it is easiest and most general to treat actions as arbitrary symbols. The user may assume any internal structure for actions (as long as τ remains invisible). For instance, when modelling transition fusion of coloured Petri nets, it may be useful to assume that actions contain tuples of data values.

Another reason for the lack of use of process-algebraic semantic models is that as such, compositional LTS construction often fails because of the spurious behaviour problem. People may have tried the basic form of compositional LTS construction, got disappointed, and rejected process algebras.

Again, this issue is not specific to process algebras but inherent in the compositional construction of behaviours. Interface processes help a lot, but they require making and modelling guesses about the behaviours of subsystems. So

their use is not fully automatic, reducing their attraction. Furthermore, it may be that the wide applicability of interface processes is not widely known. In any case, more case studies are needed to find out if interface processes are a sufficient solution.

It seems to the present author that current verification methods can process nontrivial systems, so the methods are useful, but they can process systems of industrial size only occasionally, so the methods do not meet the needs and are thus not used a lot. This holds for both process-algebraic compositional methods and verification methods in general.

In this tutorial we concentrated on the CFFD semantics. The reason is that often either it or a closely related semantics is very good for a task. If livelocks are not interesting but deadlocks are, throw divergences and infinite traces away but keep traces and stable failures. If deadlocks are not interesting but livelocks are, throw stable failures away but keep the rest. The main semantics of CSP can be used if it does not matter that there divergence is catastrophic. If branching-time properties are needed, then CFFD and CSP cannot be used, but some variant of weak or branching bisimilarity may be suitable.

We pointed out that use of a variable is essentially the same thing as parallel composition with it, and unfolding a variable can be postponed until the components of the system have been put together. They can often be put together in a stepwise manner in many different orderings. These open up possibilities for new research, to develop verification methods that apply to systems with variables but do not suffer from the effect that unfolding has to the size of the state space. Such methods must be capable of combining data manipulation steps from successive transitions, to liberate τ-transitions from data manipulation and thus make it possible to remove them in reductions. Also management of variable names is necessary, because data moves from one component to another, so names of local variables within components are not helpful in projected views. This problem was solved manually when drawing Fig. 6.

Because of networked systems and multi-core processors, today there is more need than ever to teach students basic facts about concurrency. In particular, it is important to make them realize how things may go wrong. Perhaps projected views such as in Fig. 6 and 8 can be used for that purpose.

Acknowledgements. The comments by the anonymous reviewers helped to improve this tutorial.

References

1. Arnold, A.: Finite Transition Systems. Prentice-Hall, Englewood Cliffs (1994)
2. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. Communications of the ACM 12(5), 260–261 (1969)
3. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14(1), 25–59 (1987)

4. Cleaveland, R., Hennessy, M.: Testing Equivalence as a Bisimulation Equivalence. Formal Aspects of Computing 5(1), 1–20 (1993)
5. De Nicola, R., Vaandrager, F.: Three Logics for Branching Bisimulation. Journal of the ACM 42(2), 458–487 (1995)
6. Emerson, E.A.: Temporal and Modal Logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 995–1072. The MIT Press/Elsevier (1990)
7. Engelfriet, J.: Determinacy $\longrightarrow$ (Observation Equivalence = Trace Equivalence). Theoretical Computer Science 36(1), 21–25 (1985)
8. Fernandez, J.-C.: An Implementation of an Efficient Algorithm for Bisimulation Equivalence. Science of Computer Programming 13(2-3), 219–236 (1989/1990)
9. Graf, S., Steffen, B., Lüttgen, G.: Compositional Minimisation of Finite State Systems Using Interface Specifications. Formal Aspects of Computing 8(5), 607–616 (1996)
10. Hansen, H., Valmari, A.: Operational Determinism and Fast Algorithms. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 188–202. Springer, Heidelberg (2006)
11. Hansen, H., Virtanen, H., Valmari, A.: Merging State-based and Action-based Verification. In: Lilius, J., Balarin, F., Machado, R.J. (ed.) 3rd International Conference on Application of Concurrency to System Design (ACSD 2003), pp. 150–156. IEEE Computer Society (2003)
12. Helovuo, J., Valmari, A.: Checking for CFFD-Preorder with Tester Processes. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 283–298. Springer, Heidelberg (2000)
13. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
14. Kaivola, R., Valmari, A.: The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic. In: Cleaveland, R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 207–221. Springer, Heidelberg (1992)
15. Kanellakis, P.C., Smolka, S.A.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. Information and Computation 86(1), 43–68 (1990)
16. Kangas, A., Valmari, A.: Verification with the Undefined: A New Look. In: Arts, T., Fokkink, W. (eds.) Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2003). Electronic Notes in Theoretical Computer Science, vol. 80, pp. 124–139 (2003)
17. Karsisto, K.: A New Parallel Composition Operator for Verification Tools. Dr.Tech. Thesis, Tampere University of Technology Publications 420, Tampere, Finland (2003)
18. Madelaine, E., Vergamini, D.: AUTO: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks. In: Vuong, S.T. (ed.) Formal Description Techniques II (FORTE 1989), pp. 61–66. North-Holland (1990)
19. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems, vol. I: Specification. Springer (1992)
20. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
21. Puhakka, A.: Weakest Congruences, Fairness and Compositional Process-Algebraic Verification. Dr.Tech. Thesis, Tampere University of Technology Publications 468, Tampere, Finland (2004)

22. Puhakka, A., Valmari, A.: Liveness and Fairness in Process-Algebraic Verification. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 202–217. Springer, Heidelberg (2001)
23. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
24. Roscoe, A.W.: Seeing Beyond Divergence. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) CSP25. LNCS, vol. 3525, pp. 15–35. Springer, Heidelberg (2005)
25. Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)
26. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through Determinism. Journal of Computer Security 4(1), 27–54 (1996)
27. Sabnani, K.K., Lapone, A.M., Uyar, M.Ü.: An Algorithmic Procedure for Checking Safety Properties of Protocols. IEEE Transactions on Communications 37(9), 940–948 (1989)
28. Valmari, A.: The Weakest Deadlock-Preserving Congruence. Information Processing Letters 53(6), 341–346 (1995)
29. Valmari, A.: Failure-Based Equivalences Are Faster Than Many Believe. In: Desel, J. (ed.) Structures in Concurrency Theory 1995. Workshops in Computing, pp. 326–340. Springer (1995)
30. Valmari, A.: Compositionality in State Space Verification Methods. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 29–56. Springer, Heidelberg (1996)
31. Valmari, A.: Stubborn Set Methods for Process Algebras. In: Peled, D.A., Pratt, V.R., Holzmann, G.J. (eds.) Partial Order Methods in Verification: DIMACS Workshop. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29, pp. 213–231. American Mathematical Society (1997)
32. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
33. Valmari, A.: A Chaos-Free Failures Divergences Semantics with Applications to Verification. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare, pp. 365–382. Palgrave (2000)
34. Valmari, A.: Composition and Abstraction. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 58–98. Springer, Heidelberg (2001)
35. Valmari, A.: Simple Bisimilarity Minimization in $O(m \log n)$ Time. Fundamenta Informaticae 105(3), 319–339 (2010)
36. Valmari, A., Kervinen, A.: Alphabet-Based Synchronisation is Exponentially Cheaper. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 161–176. Springer, Heidelberg (2002)
37. Valmari, A., Kokkarinen, I.: Unbounded Verification Results by Finite-State Compositional Techniques: 10^{any} States and Beyond. In: Lavagno, L., Reisig, W. (eds.) International Conference on Application of Concurrency to System Design (ACSD 1998), pp. 75–85. IEEE Computer Society (1998)
38. Valmari, A., Setälä, M.: Visual Verification of Safety and Liveness. In: Gaudel, M.-C., Woodcock, J. (eds.) FME 1996. LNCS, vol. 1051, pp. 228–247. Springer, Heidelberg (1996)
39. Valmari, A., Tienari, M.: An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm. In: Jonsson, B., Parrow, J., Pehrson, B. (eds.) Protocol Specification, Testing and Verification XI, pp. 3–18. North-Holland (1991)

40. Valmari, A., Tienari, M.: Compositional Failure-Based Semantic Models for Basic LOTOS. Formal Aspects of Computing 7(4), 440–468 (1995)
41. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM 43(3), 555–600 (1996)
42. Voorhoeve, M., Mauw, S.: Impossible Futures and Determinism. Information Processing Letters 80(1), 51–58 (2001)
43. Wolper, P., Lovinfosse, V.: Verifying Properties of Large Sets of Processes with Network Invariants. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990)

The Synthesis Problem

Wolfgang Reisig

Humboldt-Universität zu Berlin

Abstract. *Synthesis* refers to the problems of constructing a distributed system model from the sequential observation of its behaviour. This contribution suggests Petri nets as an adequate framework for solving the synthesis problem. The theory of *regions* is paramount in this context.

We present the basics of region theory and synthesis construction, and apply it to two case studies.

Introduction

A system is often modeled by a description of its observable behaviour, that is in the framework of global states and steps. To implement a system, it is often more convenient to identify *local* state components and actions whose cause and effect are limited to a few state components. This observation causes the *synthesis problem* , i.e. the problem to *synthesize* a system with local state and local action components from the description of its globally observed behaviour.

In technical terms, given a state machine, i.e. an arc labelled graph G without any assumptions about its nodes, the problem is to synthesize a Petri net N with a reachability graph isomorphic to G.

In intuitive terms, the problem is to squeeze concurrency out of sequential observations, i.e. to identify local state components within amorphic global states.

In the framework of Petri nets, this problem can be solved by the help of *region theory*. In fact, region theory belongs to the "crown jewels" of Petri nets. It provides strong indications that the basic concepts of Petri nets have been chosen most adequately to cope with current systems.

This paper has three parts: The first one sets the stage, with the illuminating example of the light/fan system, and a detailed explanation of the synthesis problem in general. The second part presents the theory of *regions*, which is the basis for all solutions of the synthesis problem. Solutions in the classes of 1-bounded Petri nets with loops allowed, are discussed in detail. The third part, finally, considers two case studies: The above mentioned light/fan system, and the well known counter flow pipeline processor, CFPP.

1 Layout of the Synthesis Problem

1.1 An Illuminating Example: The Light/Fan System

The reader is probably familiar with the common connection between lighting and air ventilation in (windowless) bathrooms: If the light is switched on while

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 300–313, 2013.

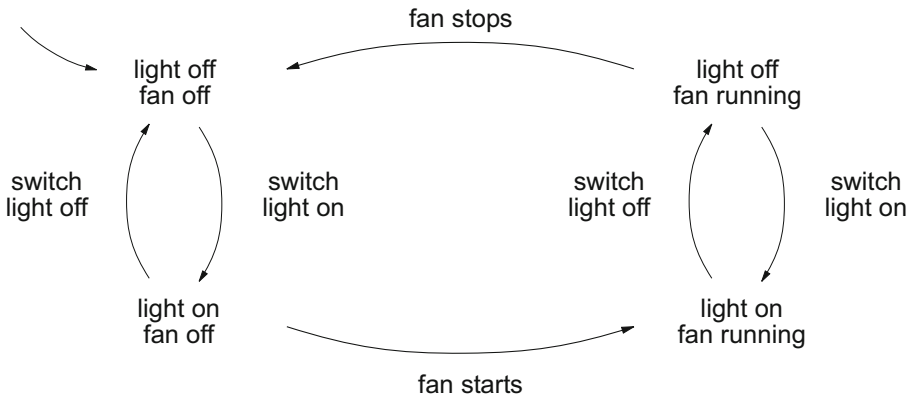


Fig. 1. The light/fan system as a state automaton

the fan is off, the latter will start as well after a while. If the light is then switched off, the fan will continue running for some time. If the fan is off and the light is first switched on and then quickly switched off again, the fan will not start at all. If the fan is running and the light is switched off and then quickly switched on again, the fan will continue running without interruption.

Traditionally, systems are often modeled as *state automata*: A state automaton consists of *states* and *steps*. One state is the *initial state*. Every step transforms one state into another and thereby executes an action. Several steps may quite well execute one and the same *action*. Technically, a state automaton can be described as a directed graph; with states as nodes and steps as labeled edges.

Figure 1 shows the behavior of the light/fan system as a state automaton Z. It has four global states and four actions, two of which (switch light on and switch light off) can occur in two states each.

Figure 2 shows the system as a Petri net N. It has four places describing the *local* states as well as four transitions, one for each action of the system.

The representation as a Petri net clearly describes the cause and effect of each action. switch light off, for example, can only occur if the light is on. The current state of the fan is irrelevant for this action. The fan itself, however, only starts if it is not running and the light is on at the same time.

It is easy to determine that the Petri net N has the exact same behavior as the state automaton Z: One constructs the reachability graph of N and asserts that it is identical to Z.

1.2 The General Question of the Synthesis Problem

Each state of the light/fan state automaton in Fig. 1 is labeled with two conditions. These conditions form places in Fig. 2. In general, however, there is no information on the states of a state automaton Z. It is *abstract*, as in Fig. 3. Every edge of such an automaton is labeled with an *action*. The same action may occur on more than one edge (this applies to a and b in Fig. 3).

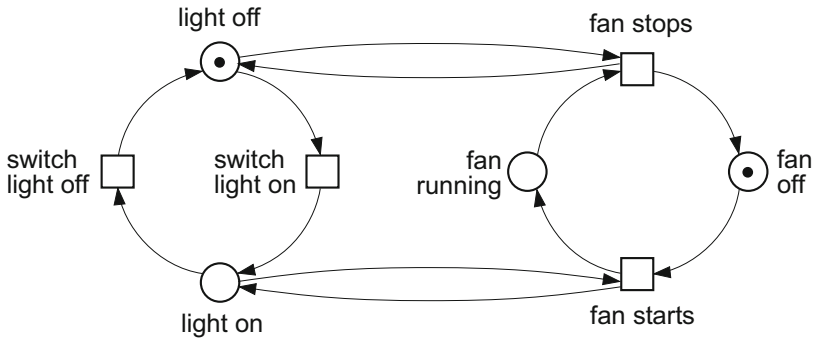


Fig. 2. The light/fan system as a Petri net

The *synthesis problem* for any (abstract) state automaton Z asks for a *distributed system* V that behaves exactly like Z. Instead of global, only *local* states may appear in V. Every action appearing in Z has to be described completely and unambiguously through its effect on a few local states in V. Here we concentrate on *Petri nets* as candidates for V. A Petri net N *behaves like* Z iff its reachability graph G is isomorphic to Z. G is isomorphic to Z if every node k of G maps to exactly one node k' of Z such that:

- the initial marking of G is mapped to the initial state of Z,
- $h \xrightarrow{t} k$ is a step in G iff $h' \xrightarrow{t} k'$ is an edge in Z.

Summarized: A Petri net N solves the synthesis problem for a given (abstract) state automaton Z if the reachability graph G of N is isomorphic to Z.

In this contribution we are behind methods to construct solutions N for the synthesis problem of given state automata Z. Those methods decisively depend on the class of nets which we conceive as candidates for N. The simplest method constructs 1-bounded, loop free nets. It succeeds, for example, for the automaton in Fig. 3, but not for the automaton L in Fig. 4. Slightly more involved is a method to construct 1-bounded nets that may include loops. It succeeds, for example, in Fig. 4 for L, but not for R. (In fact, the synthesis problem of R has no Petri net solution at all). In this contribution we stick to the construction of

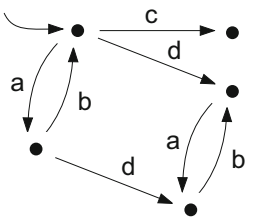
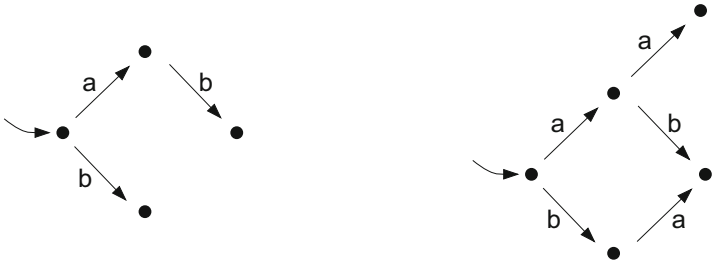


Fig. 3. state automaton Z_1



State automaton L State automaton R

Fig. 4. The synthesis problem of L is solvable, the one of R is not solvable

1-bounded solutions that may involve loops. Methods to construct n-bounded, unbounded, arc weighted, etc. solutions are behind the scope of this contribution.

2 Regions and the Synthesis of Petri Nets

2.1 Regions of State Automata

Here we define the technical construct of *regions* of state automata, Z. Intuitively formulated, a subset R of nodes is a region of Z if R "harmonizes" with the labels of the arcs that reach or leave R. In formal terms, let

$$\pi : h \xrightarrow{t} k$$

be an edge of Z ("t-edge"). We define:

R *receives* π, if $h \notin R$ and $k \in R$
R *dispatches* π, if $h \in R$ and $k \notin R$
R *contains* π, if $h \in R$ and $k \in R$

Figure 5 outlines this definition. As a region, R "harmonizes" with the labels of Z.

R is a *region of* Z if for each edge label t of Z:

- R receives either each or no t-edge and
- R dispatches either each or no t-edge.

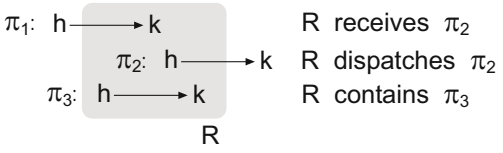
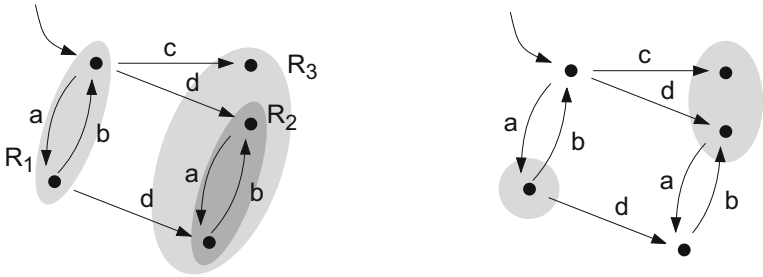


Fig. 5. Received, dispatched, and contained edges



regions of Z_1 no regions of Z_1

Fig. 6. Three regions of the state automaton Z_1: R_1 and R_2 are minimal, R_3 is not

A region R of Z is *minimal* if no proper subset of R is a region of Z.

Figure 6 shows examples and counterexamples of regions of the state automaton Z_1 in Fig. 3. Furthermore, Fig. 7 shows the minimal regions of Z_1.

2.2 The Regions of a Reachability Graph

Figure 8 shows a 1-bounded Petri net N together with its reachability graph G. As a shorthand, each marking M of G is written p or pq, representing the places p and q that carry a token. For each place p of N, let $\hat{p}$ denote the set of markings of G, with p carrying a token. One immediately observes:

$$\text{Each } \hat{p} \text{ is a minimal region of } G, \tag{2.1}$$

and vice versa:

$$\text{Each minimal region of } G \text{ is } \hat{p} \text{ for some place } p \text{ of } N. \tag{2.2}$$

It is easy to see that both properties (2.1) and (2.2) hold for *each* live, 1-bounded Petri net N, because for each transition t and each place p of N holds:

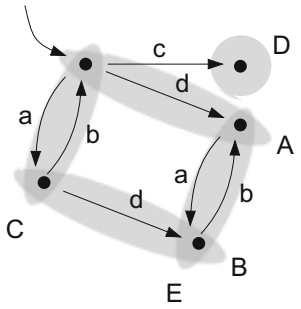


Fig. 7. The minimal regions of Z_1

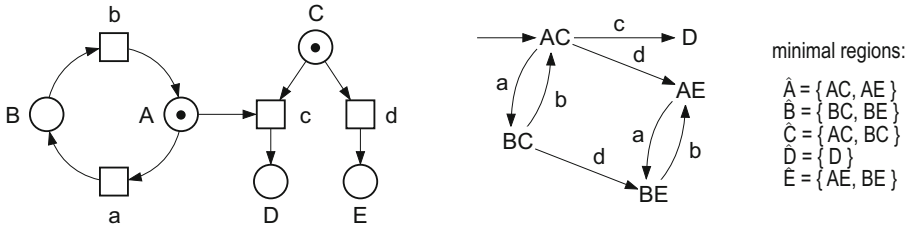


Fig. 8. A Petri net and its reachability graph with its four minimal regions $\hat{\mathsf{A}}, \ldots, \hat{\mathsf{E}}$

$$t \in {}^\bullet p \text{ in } N \text{ iff } \hat{p} \text{ receives } t \text{ in } G, \tag{2.3}$$

$$t \in p^\bullet \text{ in } N \text{ iff } \hat{p} \text{ dispatches } t \text{ in } G. \tag{2.4}$$

Solutions to the synthesis problem just exploit this observation: Given a state automaton Z, construct a place for each minimal region of Z, and a transition t for each arc label. Then link places and transitions according to (2.3) and (2.4).

2.3 The Petri Net of a State Automaton

As indicated above, each state automaton Z is assigned a Petri net N according to the following procedure:

- the minimal regions p of Z are the places of N;
- the edge labels t occurring in Z are the transitions of N;
- if the region p receives the t-edges of Z, then (t, p) is an arc of N;
- if the region p dispatches the t-edges of Z, then (p, t) is an arc of N;
- if the region p contains all the t-edges of Z, then (t, p) and (p, t) are arcs of N;
- if the initial state of Z lies within a region p of Z, the place p of N holds initially a token.

The state automaton Z_1 in Fig. 3 with its minimal regions as depicted in Fig. 7 yields the petri net of Fig. 8.

As a further example, the state automaton L in Fig. 4 has four regions. One of them contains the only - hence all - a edges. That's why the Petri net N_{L} of L, depicted in Fig. 9, contains a loop between place A and transition a.

2.4 The Synthesis Theorem

We are now ready to state the core result of this contribution: If the synthesis problem of a state automaton is solvable, then the assigned Petri net is a solution.

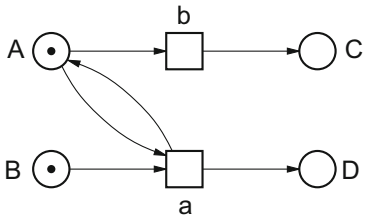


Fig. 9. Petri net N_{L} of L

Theorem 1 (synthesis theorem). *If the synthesis problem for a state automaton Z can be solved by a 1-bounded Petri net, then the assigned Petri net of Z is a solution.*

With this, it is possible to solve the general synthesis problem: One constructs the assigned Petri net N of a given state automaton Z and from this the reachability graph G of N. If Z and G are isomorphic, N obviously solves the synthesis problem for Z. Otherwise, the synthesis problem for Z cannot be solved by any 1-bounded Petri net.

Proof of this Theorem is not too difficult: It is obvious that each edge label of Z becomes a transition of the solution N of the synthesis problem of Z. Furthermore, each region of Z is obviously a candidate for a place of N. A place p corresponding to a non-minimal region is redundant: Skipping redundant places retains the structure of the reachability graph. Further details on this proof can be found in [1].

As a variant, one may be interested in loop free, 1-bounded solutions only. Then one just refrains from constructing edges (t, p) and (p, t) in case the region p contains the edge label t. For some cases (including e.g. the state automaton L of Fig. 4) the resulting net's reachability graph is not isomorphic to Z, hence the net does not solve the synthesis problem of Z.

The above procedure also fails for the the state automaton Z_2 of Fig. 10. In fact, Z_2 has no 1-bounded solution. But Z_2 has a 2-bounded solution, also shown in Fig. 10. A synthesis procedure to construct such (and much more general) solutions can be found in the literature (cf. [1]).

state automaton Z_2 2-bounded solution

Fig. 10. A synthesis problem with a 2-bounded solution

3 Case Studies

Section 3.1 solves the synthesis problem of the light/fan system as discussed in Sect. 1.1. The rest of this part discusses the synthesis of a concurrent model for the well-known counter flow pipeline processor CFPP.

3.1 The Synthesis Problem for the Light/Fan State Automaton

To simplify the argumentation, we use an abridged version of the denotations of the light/fan system, as shown in Fig. 11.

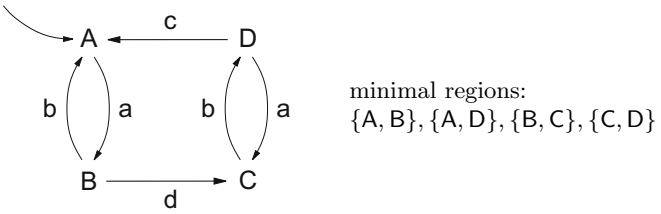


Fig. 11. Abridgment of Fig. 1

This state automaton has four minimal regions. According to the procedure explained above, the system net in Fig. 12 is constructed.

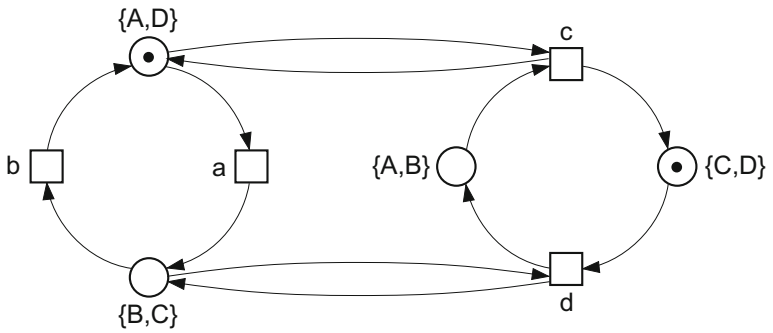


Fig. 12. Solution to the synthesis problem for the state automaton in Fig. 11

The construction of its reachability graph G is left to the reader, as well as the ascertainment that G is isomorphic to Fig. 11. With this, Fig. 12 solves the synthesis problem for the state automaton in Fig. 11. Changing the abridged denotations in Fig. 12 back to their longer versions, in fact, results in the system net shown in Fig. 2.

3.2 The Counterflow Pipeline Processor (CFPP): The Problem

The Sprout counterflow pipeline processor is a well-known, complex asynchronous hardware architecture. It utilizes as sequence $P = M_1 \dots M_k$ of consecutively linked modules, as outlined in Fig. 13.

Data packets $\mathsf{d}_1, \dots, \mathsf{d}_n$ flow from the left, that is via M_1, into P. The computed data packets $\mathsf{e}_1, \dots, \mathsf{e}_n$ leave P on the right, via M_k. Vice versa, instructions flow from the right into P and leave P via M_1. All modules work according to the same pattern. A module M_i can receive data packets from its left neighbor M_{i-1} and give out instructions to it. To its right neighbor M_{i+1}, it can give out data packets and receive instructions from it. The modules communicate synchronously: M_i can give out a data packet to M_{i+1} as M_{i+1} receives it. In analogy, M_i gives out an instruction to M_{i-1} as M_{i-1} receives it.

Figure 14 shows the behavior of an "inner" module $\mathsf{M}_i (i = 2, \dots, k-1)$ as a state automaton.

Initially, M_i can receive a data packet d (from M_{i-1}) and and instruction f (from M_{i+1}). When M_i has received both – in arbitrary order – it is ready for computation and applies f to d. M_i can then – in arbitrary order – give out the

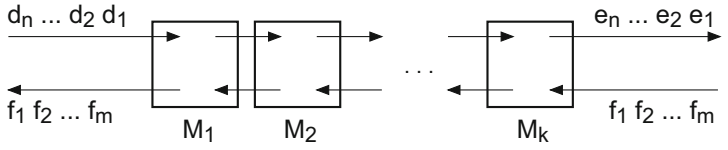


Fig. 13. Assembly of the CFPP from modules M_i

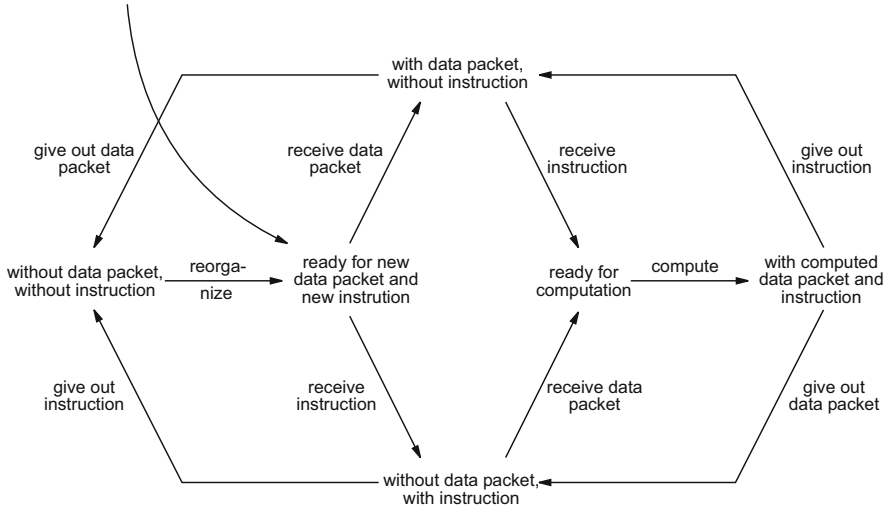
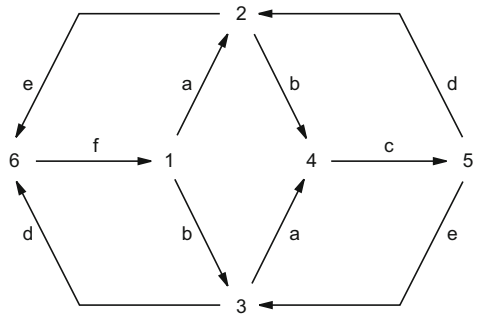


Fig. 14. A CFPP module as state automaton



$A = \{ 2, 4, 5 \}$, $B = \{ 3, 4, 5 \}$,
$C = \{ 1, 3, 6 \}$, $D = \{ 1, 2, 6 \}$,
$E = \{ 1, 3, 5 \}$, $F = \{ 2, 4, 6 \}$,
$G = \{ 5, 1, 2 \}$, $H = \{ 3, 4, 6 \}$

Fig. 15. The 8 minimal regions $A, \ldots, H$ of a CFPP module

newly computed data packet and the instruction to M_{i+1} and M_{i-1} respectively. M_i does not store anything then, reorganizes itself and returns to its initial state.

Of particular interest are the two states (top and bottom in Fig. 14) in which M_i stores either only a data packet or only an instruction: M_i can give out the unprocessed data packet to M_{i+1}, or respectively the unused instruction to M_{i-1}.

The module M_1 on the left edge of the CFPP architecture essentially behaves like an inner module.However, it only gives out an instruction f_i to the environment after f_i has been applied to the last data packet d_n. In analogy, the module M_k on the right edge only gives out a data packet e_i to the environment after the last instruction f_m has been applied to e_i. We do not explicitly model those two modules.

An architecture with k modules can process a stream $\mathsf{d}_1, \ldots, \mathsf{d}_n$ of data packets and a stream $\mathsf{f}_1, \ldots, \mathsf{f}_m$ of instructions if and only if n and m are not both greater than k: In that case, the data packets or instructions can all be stored simultaneously inside the modules. A CFPP can compensate for the different durations of instructions only if k is greater than either n or m. For further details we refer to [8].

3.3 The Synthesis Problem for the CFPP

The state automaton in Fig. 14 uses six different actions, four of which can occur in two states each. We now ask for the pre-, post- and side conditions of their occurrences and thus for the local states that organize a CFPP module. For this purpose, we solve the synthesis problem for the state automaton of Fig. 14. Using the denotations in Fig. 15, its eight minimal regions generate the Petri net N shown in Fig. 16. Its reachability graph is isomorphic to the state automaton in Fig. 14. Therefore, N solves the synthesis problem for the CFPP.

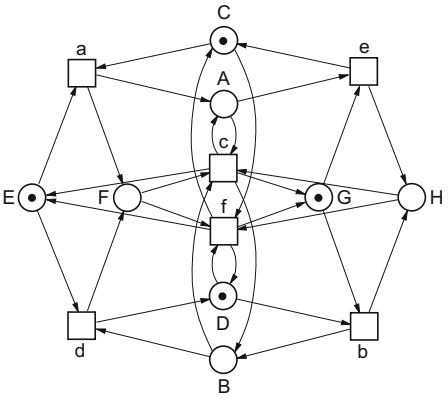


Fig. 16. The solution to the synthesis problem for the CFPP

3.4 Structural Simplification of a Module

The structure of the Petri net N in Fig. 16 can be simplified. At first, we derive for this:

$$\begin{aligned} \mathsf{A}+\mathsf{H}+\mathsf{E}+\mathsf{D} &= 2 && \text{place invariant}\\ \mathsf{E}+\mathsf{F} &= 1 && \text{place invariant}\\ -\mathsf{B}-\mathsf{D} &= -1 && \text{place invariant}\\ -2\mathsf{E} &\leq 0 && \text{canonical inequality of } \mathsf{E} \end{aligned}$$

The combination of these yields:

$$\mathsf{A}+\mathsf{F}+\mathsf{H}-\mathsf{B} \leq 2.$$

From this follows for each reachable marking that marks A, F and H, that it also marks B. This renders the loop between B and c redundant.

The argumentation about the loop between C and f is analogous. With this, Fig. 17 shows the final version of a CFPP module.

3.5 The Model of the CFPP

To form the CFPP P as a sequence $\mathsf{M}_1 \ldots \mathsf{M}_k$ of modules, k instances of the module in Fig. 17 have to be combined. It is rather simple to combine a module M_i with its right neighbor M_{i+1}: The transition e of M_i is identified with the transition a of M_{i+1} and, in analogy, b of M_i with d of M_{i+1}. Figure 18 outlines this construction.

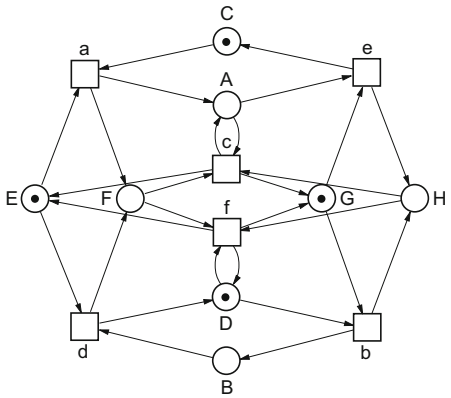


Fig. 17. Final version of a module

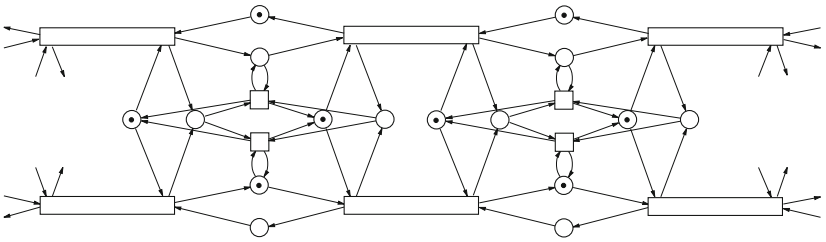


Fig. 18. Combination of modules M_i and M_{i+1} of the CFPP

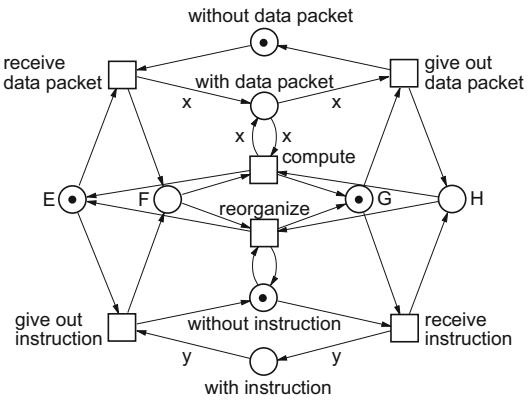


Fig. 19. Module of the CFPP

Further Reading

The solution to the synthesis problem belongs to those success stories in the field of Petri nets that asserted themselves only slowly. By now, however, it belongs – in various versions – to the standard repertoire of most Petri net analysis tools.

Already in the late 1970s, Carl Adam Petri knew that is is possible to back calculate any 1-bounded Petri net N from its reachability graph G: A state set A of G forms a place in N if either all or no edges of G that have the same label either start or end in A. He did not consider that particularly interesting, because n places of N would result in 2^n nodes in G and thus 2^{2^n} state sets – an unmanageable situation.

In the mid-1980s, Ehrenfeucht and Rozenberg [4] solved the synthesis problem for loop-free, 1-bounded Petri nets as an example of use for their "2-Theory". For this, they use *all* the regions of G for the construction of a system net. It was not until 1993 that Bernardinello [1] confirmed the assumption that the minimal regions suffice. Only with this, region theory became usable in practice. By using only the minimal regions, nets of manageable sizes are constructed.

Many authors took part in generalizing the 1-boundedness in Theorem 1 to more general Petri nets and in including even inhibitor edges. With this, the synthesis problem of much more general state automata becomes solvable. The most important authors include Baudouel, Bergenthum, Busi, Cordella, Darondeau, Desel, Gunther, Hoagens, Juhas, Kishinevsky, Kleijn, Kontratyev, Lavagno, Lorenz, Mauser, Mukund, Pietkiewicz-Koutny, Pinna, Thiagarajan, van der Aalst and Yakovlev. An overview of these developments can be found in [5]. Also very interesting for practical purposes is a weakening of the criteria for the solution to the synthesis problem: A system net N solves a weak synthesis problem for a state automaton Z if the reachability graph G of N is not necessarily isomorphic to Z, but if G and Z are bisimular or fulfill some other simulation relation. In [2], for instance, only a bisimulation is required between the state automaton Z and the reachability graph of the synthesized net N. Additionally, the transitions of N may be labeled. With this, one action of Z can be implemented by several transitions of N. Lastly, it is possible to require special characteristics of N, for instance the free-choice characteristic. [7] solves the synthesis problem for other, liberal simulation relations and also takes distributed runs into account. Numerous software tools for the analysis of Petri nets offer a synthesis module, for instance the tool Petrify [3].

Put intuitively, the synthesis problem seeks to construct a distributed system from the observance of its sequential behavior. The fact that this is possible marks Petri nets as a "very natural modeling technique" for distributed systems.

References

1. Bernardinello, L.: Synthesis of Net Systems. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 89–105. Springer, Heidelberg (1993)
2. Cortadella, J., Kishinevsky, M., Lavagne, L., Yakovlev, A.: Deriving Petri Nets from Finite Transition Systems. IEEE Transactions on Computers 47, S.859–S.882 (1998)
3. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. IEICE Transactions on Information and Systems E 80-D(3), 315–325 (1997)
4. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures. Acta Inf. 27, S.315–S.368 (1990)
5. Lorenz, R., Mauser, S., Juhas, G.: How to synthesize nets from languages: a survey. In: 39th Winter Simulation Conference, pp. 637–647. IEEE Press (2007)
6. Reisig, W.: Petri Nets. Springer (2010) (to appear) German Version: Petri Netze. Vieweg+Teubner
7. Vogler, W.: Concurrent implementation of asynchronous transition systems. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 284–303. Springer, Heidelberg (1999)
8. Yakovlev, A.V., Koelmans, A.M.: Petri nets and digital hardware design. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1492, pp. 154–236. Springer, Heidelberg (1998)

Models from Scenarios

Robert Lorenz[1,⋆], Jörg Desel[2], and Gabriel Juhás[3]

[1] Department of Computer Science
University of Augsburg, Germany
robert.lorenz@informatik.uni-augsburg.de
[2] Department of Software Engineering
Distance University of Hagen, Germany
joerg.desel@fernuni-hagen.de
[3] Faculty of Electrical Engineering and Information Technology
Slovak University of Technology, Bratislava, Slovakia
gabriel.juhas@stuba.sk

Abstract. Synthesis of Petri nets from behavioral descriptions has important applications in the design of systems in different application areas. In this paper we present a survey on the technique of region based synthesis of Petri nets from languages. Each word in a given language specifies one run of the searched Petri net, i.e. represents one observable scenario of the system.

We concentrate on recent developments for languages of different kinds of causal structures (such as partial orders and stratified order structures). Causal structures represent causal relationships between events of one run. Expressible causal relationships are for example direct and indirect causal dependency, concurrency and synchronicity of events.

Concerning infinite languages, several possibilities of a finite representation are discussed. As the goal of synthesis, place/transition nets and inhibitor nets as well as several restrictions of these net classes are used. The presented framework integrates all classical results on sequential languages.

Keywords: Synthesis, Region Theory, Petri Net, Causal Semantics, Partial Language, Partial Order, Stratified Order Structure.

1 Introduction

Synthesis of Petri nets from behavioral descriptions has been a successful line of research since the 1990s. There is a rich body of nontrivial theoretical results and there are important applications in industry, in particular in hardware design [9,19], in control of manufacturing systems [33] and recently also in process mining [32,31,4,17] and workflow design [12,6].

The synthesis problem is the problem to construct, for a given behavioral specification, a Petri net such that the behavior of this net coincides with the specified behavior (if such a net exists). There are many different methods which are presented in literature to solve this problem. They differ mainly in the Petri net class and the model for

⋆ Supported by the German Research Council, project SYNOPS 2008 - 2012.

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 314–371, 2013.

the behavioral specification considered. All these methods are based on one common theoretical concept, the notion of a *region* of the given behavioral specification.

In this paper, we present an overview of region-based synthesis methods, which regard languages as behavioral specifications, where each word in a given language specifies one run of the searched Petri net. Classical results consider sequential languages representing sequential runs of Petri nets. Recent developments examine languages of different kinds of causal structures (such as partial orders and stratified order structures) representing non-sequential runs. Such causal structures are able to represent different causal relationships between events of one run, such as for example direct and indirect causal dependency, concurrency and synchronicity.

In the following we describe the general approach of region based synthesis from languages. Denote the set of runs of a Petri net N by $L(N)$. It depends on the Petri net class and the considered net semantics, which kind of runs are considered in $L(N)$. Formally the synthesis problem w.r.t. different Petri net classes and different language types is:

Given: A prefix-closed language L over a finite alphabet of transition names T.
Searched: A Petri net N with set of transitions T and $L(N) = L$.

This means, we search for an exact solution of the problem. Such an exact a solution may not exist, i.e. not each language L is a *net language*.

The classical idea of region-based synthesis is as follows: First consider the net N having an empty set of places but all transitions occurring as labels in L. This net generates each execution in L (i.e. $L \subseteq L(N)$), because there are no places restricting transition occurrences. But it generates much more executions. Since we are interested in an exact solution, we restrict $L(N)$ by adding places.

There are places p, which restrict the set of executions too much in the sense that $L \setminus L(N) \neq \emptyset$, if p together with adjacent weighted arcs is added to N. Such places are called *non-feasible (w.r.t. L)*. We only add so called *feasible* places p satisfying $L \subseteq L(N)$, if p is added to N (Figure 1). The idea of region-based synthesis is to add *all* feasible places to N. The resulting net N_{sat} is called the *saturated feasible net*. N_{sat} has by construction the following very nice property:

(min) $L(N_{sat})$ is the smallest net language satisfying $L \subseteq L(N_{sat})$.

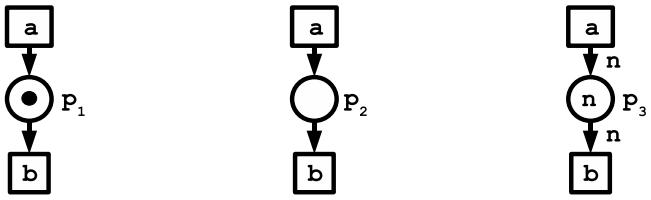


Fig. 1. The place p_1 is feasible, the place p_2 is not feasible w.r.t. the language $L = \{a, b, ab, ba, abb, bab\}$ (b is no execution of the net shown in the middle). The place p_3 is feasible w.r.t. L for each integer $n \in \mathbb{N}$

This is clear, since $L(N_{sat})$ could only be further restricted by adding non-feasible places. The property (min) directly implies that there is an exact solution of the synthesis problem if and only if N_{sat} is such an exact solution. Moreover, if there is no exact solution, N_{sat} is the best approximation to such a solution "from above".

Unfortunately, this result is only of theoretical value, since the set of feasible places is in general *infinite* (Figure 1). Therefore, for a practical solution, a finite subset of the set of all feasible places is defined, such that the net N_{fin} defined by this finite subset fulfills $L(N_{fin}) = L(N_{sat})$. Such a net N_{fin} is called *finite representation* of N_{sat}. In order to construct such a finite representation, in an intermediate step a feasible place is defined through a so called *region* of the given language L, where the set of all regions equals the set of non-negative integral solutions of an appropriate linear system of the form $\mathbf{A}_L \cdot \mathbf{x} \leq \mathbf{b}_L$.

The described approach is common to all known region-based synthesis methods (see Figure 2), where different notions of regions and of finite representations N_{fin} are used. There are two types of definitions of regions and two types of definitions of finite representations, whose four combinations cover all known region-based synthesis methods. All these combinations can be applied to almost each Petri net class and each language type (leading to different nets N_{fin} having the same behavior).

Summarizing, the form of the synthesis problem and the solution method can be varied along the following lines: *Petri net class*, *language type*, *region type* and *finite representation type*. This paper presents a common framework for all these variations based on a combination of and extending the publications [27], [26] and [7].

The organisation of the paper is as follows: In the first part we develop a basic framework considering the synthesis of place/transition nets from finite and from simple infinite languages of labelled partial orders, using both region types and both finite representation types. For the finite specification of infinite languages a simple term based notation is used. In the second part we extend and generalize the basic framework along several lines:

- We consider the synthesis of inhibitor nets from finite and from simple infinite languages of labelled stratified order structures.
- We discuss synthesis from languages of non-transitive order structures.

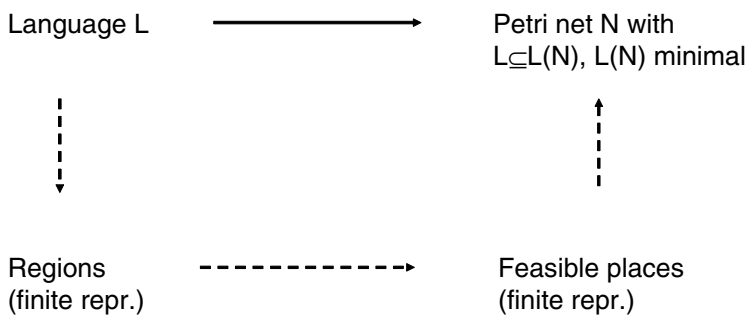


Fig. 2. The approach of region-based synthesis

- We examine the synthesis of nets of restricted net classes.
- We suggest several possibilities for a finite representation of more general infinite languages.

We do not consider labelled Petri nets (this is another line of research), and nets with invisible (internal) transitions or high level Petri nets (these are future topics of research).

This paper only gives a technical overview. Case studies and issues from practise are out of scope of this paper.

2 Basic Framework

In this section we present all main concepts by means of the synthesis of place/transition Petri nets from languages of labelled partial orders.

2.1 Mathematical Preliminiaries

In this subsection we present necessary notions and definitions including labelled partial orders, place/transition Petri nets and runs of place/transition Petri nets.

Basic Notions. By $\mathbb{N}_0$ we denote the set of *nonnegative integers*, by $\mathbb{N}$ the set of *positive integers*.

Given a function f from X to Y and a subset Z of X we write $f|_Z$ to denote the *restriction* of f to the set Z.

Given a finite set X, the symbol $|X|$ denotes the *cardinality* of X. The set of all subsets of X is denoted by $\mathcal{P}(X)$.

The set of all *multisets* over a set X is the set $\mathbb{N}^X$ of all functions $f : X \to \mathbb{N}$. Addition $+$ on multisets is defined by $(m + m')(x) = m(x) + m'(x)$. The relation $\leq$ between multiset is defined through $m \leq m' \iff \exists m''(m+m'' = m')$. We define $x \in m$ if $m(x) > 0$. A multiset is *finite*, if $\sum_{x \in X} m(x)$ is finite. A set $A \subseteq X$ is identified with the multiset m satisfying $m(x) = 1 \iff x \in A \wedge m(x) = 0 \iff x \notin A$. The support of a multiset m ist the set $set(m) = \{x \mid x \in m\}$. If X is finite, a multiset m we also write in the form of an $|X|$-tuple $(m(x)x)_{x \in X}$. For example, the finite multiset m over $\{a, b, c\}$ defined by $m(a) = 1$ and $m(b) = 2$ we denote by $(1a, 2b, 0c)$. A multiset m satisfying $m(a) > 0$ for exactly one element a we call *singleton multiset* and denote it by $m(a)a$. The multiset m satisfying $\forall x \in X : \ m(x) = 0$ we call *empty multiset* and denote it by ϵ.

Let X, T be sets and $l : X \to T$ be a labelling function assigning to each $x \in X$ a label $l(x)$ from T. Such a labelling function can be lifted to subsets $Y \subseteq X$ in the following way: $l(Y)$ is the multiset over T given by $l(Y)(t) = |l^{-1}(t) \cap Y|$.

Given a binary relation $R \subseteq X \times Y$ and a binary relation $S \subseteq Y \times Z$ for sets X, Y, Z, then their composition is defined by $R \circ S = \{(x, z) \mid \exists y((x, y) \in R \wedge (y, z) \in S)\} \subseteq X \times Z$. For a binary relation $R \subseteq X \times X$ over a set X, we denote $R^1 = R$ and $R^n = R \circ R^{n-1}$ for $n \geq 2$. The symbol R^+ denotes the *transitive closure* $\bigcup_{n \in \mathbb{N}} R^n$ of R and the symbol R^* denotes the *reflexive transitive closure* $R^+ \cup \{(x, x) \mid x \in X\}$ of R. We also write aRb to denote $(a, b) \in R$.

Let A be a finite set of characters. A *(classical) language over* A is a (possibly infinite) set of finite sequences of characters from A. For a language L and $w \in L$, $|w|_a$ denotes the number of a's occurring in w (for example $|aba|_a = 2$). A *(concurrent) step over* A is a multiset over A. A *step language over* A is a (possibly infinite) set of finite sequences of steps over A. For a sequence of steps $w = \alpha_1 \dots \alpha_m$, $|w|_a = \sum_{i=1}^{m} \alpha_i(a)$ denotes the number of a's occurring in w (for example $|(1a, 0b)(0a, 2b)|_b = 2$).

Partial Orders. A *directed graph* is a pair $G = (V, \rightarrow)$, where V is a finite *set of nodes* and $\rightarrow \subseteq V \times V$ is a binary relation over V, called the *set of edges* (all graphs considered in this paper are finite). The set of nodes of a directed graph G is also denoted by $V(G)$. The *preset* of a node $v \in V$ is the set ${}^\bullet v = \{u \mid u \rightarrow v\}$. The *postset* of a node $v \in V$ is the set $v^\bullet = \{u \mid v \rightarrow u\}$. The *preset* of a subset $W \subseteq V$ is the set ${}^\bullet W = \bigcup_{w \in W} {}^\bullet w$. The *postset* of a subset $W \subseteq V$ is the set $W^\bullet = \bigcup_{w \in W} w^\bullet$. A *path* is a sequence of (not necessarily distinct) nodes $v_1 \dots v_n$ $(n > 1)$ such that $v_i \rightarrow v_{i+1}$ for $i = 1, \dots, n-1$. A path $v_1 \dots v_n$ is a *cycle*, if $v_1 = v_n$. A directed graph is called *acyclic*, if it has no cycles. The set of maximal nodes of an acyclic directed graph $G = (V, \rightarrow)$ is the set $Max(G) = \{v \mid v^\bullet = \emptyset\}$, the set of its minimal nodes is the set $Min(G) = \{v \mid {}^\bullet v = \emptyset\}$. An acyclic directed graph $(V, \rightarrow')$ is an *extension* of an acyclic directed graph $(V, \rightarrow)$ if $\rightarrow \subseteq \rightarrow'$. An acyclic directed graph $(V', \rightarrow)$ is a *prefix* of an acyclic directed graph $(V, \rightarrow)$ if $V' \subseteq V$ and $(v' \in V') \wedge (v \rightarrow v') \Rightarrow (v \in V')$. An acyclic directed graph $(V', \rightarrow)$ is a *sub-graph* of an acyclic directed graph $(V, \rightarrow)$ if $V' = U \setminus W$ for prefixes $(U, \rightarrow)$ and $(W, \rightarrow)$. Then $(W, \rightarrow)$ is called *prefix of the sub-graph* $(V', \rightarrow)$.

A *partial order* over a set V is a binary relation $< \subseteq V \times V$ which is irreflexive $(\forall v \in V : v \not< v)$ and transitive $(<=<^+)$. We associate a finite partial order $<$ over V with the directed graph $(V, <)$.

Two nodes $v, v' \in V$ of a partial order $(V, <)$ are called *independent* if $v \not< v'$ and $v' \not< v$. By $co_< \subseteq V \times V$ we denote the set of all pairs of independent nodes of V. A *co-set* is a subset $C \subseteq V$ fulfilling $\forall x, y \in C : x \, co_< \, y$. A *cut* is a maximal co-set w.r.t. set inclusion. For a co-set C of a partial order $(V, <)$ and a node $v \in V \setminus C$ we write $v < C$, if $v < s$ for an element $s \in C$ and $v \, co_< \, C$, if $v \, co_< \, s$ for all elements $s \in C$. The sets $Max(po)$ and $Min(po)$ are cuts.

The *skeleton* of a finite partial order $po = (V, <)$ is the minimal relation $\prec \subseteq <$ satisfying $\prec^+ = <$.

Graphically, nodes of partial orders are drawn as small squares and the relatrion by (drawn-through) arrows between nodes. Figure 3 shows an example partial order po. The nodes v_1 and v_2 as well as v_3 and v_2 are independent. It holds $Max(po) = \{v_2, v_3\}$ and $Min(po) = \{v_1, v_2\}$.

Place/Transition Petri Nets. A *net* is a 3-tuple $N = (P, T, F)$, where P is a finite set of *places*, T is a finite set of *transitions* disjoint from P and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. A *marking* of a net assigns to each place $p \in P$ a number $m(p) \in \mathbb{N}_0$, i.e. a marking is a multiset over P. A *marked net* is a net $N = (P, T, F)$ together with an *initial marking* m_0. Graphically, places are drawn as circles, transitions as squares and the flow relation as arrows between places and transitions. A marking m is illustrated by drawing $m(p)$ tokens inside place p.

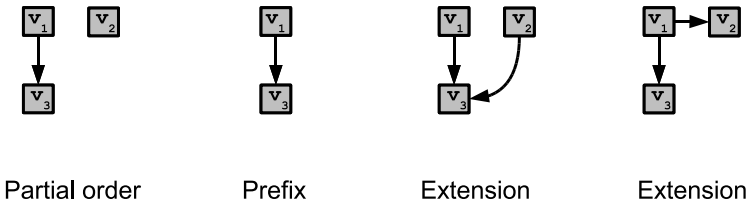


Fig. 3. Example of a partial order and a prefix and two extensions of this partial order

Definition 1 (Place/Transition Petri Net). *A* place/transition Petri net *(*PT-net*) is a 4-tuple* $N = (P, T, F, W)$*, where* (P, T, F) *is a net and* $W : (P \times T) \cup (T \times P) \to \mathbb{N}_0$ *is a* weight function *satisfying* $W(x, y) > 0 \Leftrightarrow (x, y) \in F$.

Graphically, the number $W(x, y)$ is assigned to an arrow from x to y, if $W(x, y) > 1$ (that means, $W(x, y) = 1$ for arrows (x, y) without assigned weight). Figure 4 shows a marked PT-net with $P = \{p_1, p_2, p_3\}$, $T = \{a, b\}$, $m_0(p_1) = m_0(p_2) = 1$, $m_0(p_3) = 0$ and $W(p_1, a) = W(a, p_2) = W(p_2, b) = W(b, p_3) = 1$.

We introduce the following multisets of places:

- ${}^\bullet t(p) = W(p, t)$ and $t^\bullet(p) = W(t, p)$ for transitions t.
- ${}^\bullet \tau(p) = \sum_{t \in T} \tau(t)\, {}^\bullet t(p)$ and $\tau^\bullet(p) = \sum_{t \in T} \tau(t) t^\bullet(p)$ for multisets of transitions τ.

The definition of executions of PT-nets depends on the *occurrence rule* of transitions, stating in which markings a transition (or a multiset of transitions) can occur and how these markings are changed by its occurrence.

Definition 2 (Occurrence Rule). *A transition* $t \in T$ can occur *in a marking* m*, if* $m \geq {}^\bullet t$*. A multiset of transitions* τ can occur *in* m*, if* $m \geq {}^\bullet \tau$.

If a transition t *occurs in a marking* m*, the resulting marking* m' *is defined by* $m' = m - {}^\bullet t + t^\bullet$*. If a multiset of transitions* τ *occurs in* m*, then the resulting marking* m' *is defined by* $m' = m - {}^\bullet \tau + \tau^\bullet$*. We write* $m \xrightarrow{t} m'$ *(*$m \xrightarrow{\tau} m'$*) to denote that* t *(*τ*) can occur in* m *and that its occurrence leads to* m'.

The number $W(p, t)$ represents the number of tokens *consumed from* p by an occurrence of t and the number $W(t, p)$ represents the number of tokens *produced in* p by an occurrence of t.

The occurrence of a multiset of transitions τ in a marking m means, that all transitions in τ occur in parallel.

The notion of *execution* depends on the chosen net semantics. In the following definition we consider sequential semantics and step semantics. Causal semantics is defined in the next subsection.

Definition 3 (Execution). *A* sequential execution in m *of a PT-net is a finite sequence of transitions* $\sigma = t_1 \dots t_n$ *such that there are markings* $m_1, \dots, m_n$ *satisfying* $m \xrightarrow{t_1} m_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} m_n$.

A step execution *in m of a PT-net is a finite sequence of multisets of transitions $\sigma = \tau_1 \dots \tau_n$ such that there are markings $m_1, \dots, m_n$ satisfying $m \xrightarrow{\tau_1} m_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_n} m_n$.*

We write $m \xrightarrow{\sigma} m_n$ to denote the occurrence of such executions σ.

Each sequential execution is also a step execution. The markings which can be reached from the initial marking via sequential executions (resp. step executions) are called *reachable*.

The PT-net shown in Figure 4 has the sequential executions a, b, ab, ba, abb, bab and the additional step execution $(1a, 1b)(0a, 1b), (1a, 0b)(0a, 2b)$ in the initial marking.

If τ is a multiset of transitions which can occur in a marking m and $\tau = t_1 + \ldots + t_n$ for transitions $t_1, \dots, t_n$, then $t_1 \dots t_n$ is a sequential execution in m, i.e. the transitions in τ can occur in m in arbitrary sequential order.

Finally, we recall process semantics of PT-nets.

Definition 4 (Occurrence Net). *An* occurrence net *is a net $O = (B, E, G)$ satisfying:*

- *B and E are finite and disjoint sets.*
- *$G \subseteq (B \times E) \cup (E \times B)$.*
- *$(B \cup E, G)$ is a directed acyclic graph.*
- *$\forall b \in B(|{}^\bullet b| \leq 1 \wedge |b^\bullet| \leq 1)$.*

The elements of B are called conditions *and the elements of E are called* events*. The relation G is called* flow relation*.*

Since an occurrence net can be identified with an acyclic directed graph, we use notations introduced for acyclic directed graphs also for occurrence nets. A *slice* of an occurrence net is a cut consisting solely of conditions.

In a process, the events of an occurrence net are interpreted as transition occurrences of a PT-net. Conditions represent tokens in places.

Definition 5 (Process). *Let $N = (P, T, F, W, m_0)$ be a marked PT-net. A* process *of N is a pair $K = (O, \rho)$, where $O = (B, E, G)$ is an occurrence net and $\rho : B \cup E \to P \cup T$ is a labelling function, satisfying*

- *$\rho(B) \subseteq P$ and $\rho(E) \subseteq T$.*
- *$\forall e \in E : \ \rho({}^\bullet e) = {}^\bullet \rho(e) \wedge \rho(e^\bullet) = \rho(e)^\bullet$.*
- *$\rho(Min(O)) = m_0$.*

In a process of a PT-net, two transition occurrences are *directly causally dependent* if one transition occurrence e' consumes tokens which are produced by the other transition occurrence e. Such a situation is called *token flow* between transition occurrences and can be directly observed in a process via $e^\bullet \cap {}^\bullet e' \neq \emptyset$. Figure 4 shows a process of a PT-net, where names of conditions are omitted. The names of events are shown inside, the labels of events and conditions outside of the graphical object. In this process, v_1 and v_3 are directly causally dependent.

For each slice C of a process, $\rho(C)$ is a reachable marking of the net. On the other hand, for each reachable marking m there is a slice C in some process such that $\rho(C) = m$.

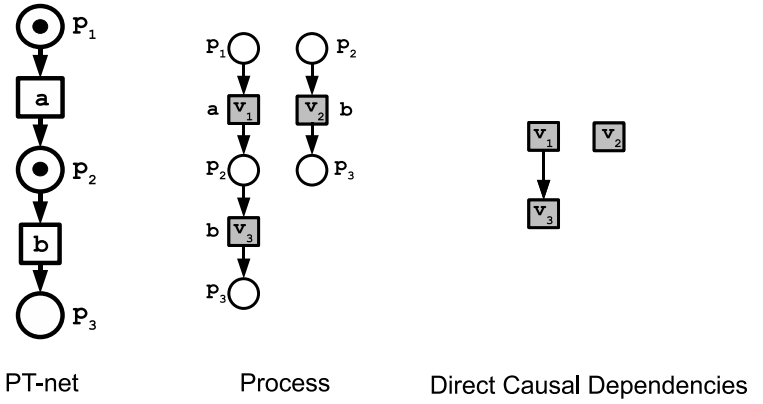


Fig. 4. Example of a PT-net and one of its processes

2.2 Causal Semantics

We use partial orders labelled by transition names to represent single (non-sequential) runs of PT-nets. The nodes of a partial order represent transition occurrences and its arrows an "earlier than"-relation between transition ocurrences in the sense that one transition occurrence can be observed earlier than another transition occurrence. If there are no arrows between two transition occurrences, then these transition occurrences are independend and are called *concurrent*. Concurrent transition occurrences can be observed in arbitrary sequential order and in parallel. This interpretation of arrows is called *occurrence interpretation*.

Definition 6 (Labelled Partial Order). *A* labelled partial order *(*LPO*)* over T *is a 3-tuple* $(V, <, l)$*, where* $(V, <)$ *is a partial order and* $l : V \to T$ *is a labelling function on* V*.*

We only consider LPOs up to isomorphism, i.e. only the labelling of events is of interest, but not the event names. Formally, two LPOs $(V, <, l)$ and $(V', <', l')$ are *isomorphic*, if there is a renaming function $I : V \to V'$ satisfying $l(v) = l'(I(v))$ and $v < w \Leftrightarrow I(v) <' I(w)$.

A *linear order* is an LPO $(V, <, l)$ where $<$ is a total order, i.e. there is no independence between transition occurrences: $\forall u, v \in V : \ u < v \vee v < u$. Linear orders represent sequential executions of Petri nets in the obvious way. For example, the LPO lpo_4 shown in Figure 5 is linear and represents the sequential execution abb.

A *stepwise linear LPO* is an LPO $(V, <, l)$ where the relation $co_<$ is transitive. The maximal sets of independent transition occurrences are called *steps*. The steps of a stepwise linear LPOs are linearly ordered. Thus, stepwise linear LPOs represent step executions of Petri nets. For example, the LPO lpo_1 shown in Figure 5 is not stepwise linear, while the LPOs lpo_2 (representing the step execution $(1a, 1b)(0a, 1b)$) and lpo_3 (representing the step execution $(1a, 0b)(0a, 2b)$) are stepwise linear.

The *set of step-linearizations* of an LPO is the set of stepwise linear LPOs which are extensions of this LPO. For example, the LPOs lpo_2 and lpo_3 shown in Figure 5 are step linearizations of lpo_1.

Definition 7 (LPO-run). *Let $N = (P, T, F, W, m_0)$ be a PT-net. An LPO $(V, <, l)$ is a* LPO-run *of N if there is a process $K = (O, \rho)$, $O = (B, E, G)$, of N such that $(V, <)$ is an extension of $(E, \{(e, f) \mid e^\bullet \cap {}^\bullet f \neq \emptyset\})$ and $l = \rho|_E$.*

An LPO-run lpo of N is said to be minimal*, if there exists no other LPO-run lpo' of N such that lpo is an extension of lpo'.*

Note that $(E, \{(e, f) \mid e^\bullet \cap {}^\bullet f \neq \emptyset\})$ is an acyclic directed graph representing all direct causal dependencies between transition occurrences of a process of the net. This means, along the "earlier than"-relations between transition occurrences of an LPO-run token flow is allowed, but not required. Figure 5 shows a PT-net together with some of its LPO-runs. Note that the LPO-run lpo_1 exactly represents all direct causal dependencies between transition occurrences of a process of the net (which is shown in Figure 4). Moreover, lpo_1 is minimal, since a second occurrence of b must be preceeded by an occurrence of a.

From the definition follows that extensions of LPO-runs also are LPO-runs. This means, the set of all LPO-runs can be deduced from the set of minimal LPO-runs.

There are two alternative but equivalent definitions of LPO-runs in literature:

- An LPO $lpo = (V, <, l)$ is an LPO-run of a PT-net N if and only if each step-linearization of lpo is a step execution of N. This means, LPO-runs are consistent with the step semantics of PT-nets.
- An LPO $lpo = (V, <, l)$ is an LPO-run if and only if for each cut C of lpo and each place p there holds:

$$m_0(p) + \sum_{v<C} (W(l(v), p) - W(p, l(v))) \geq \sum_{v \in C} W(p, l(v)).$$

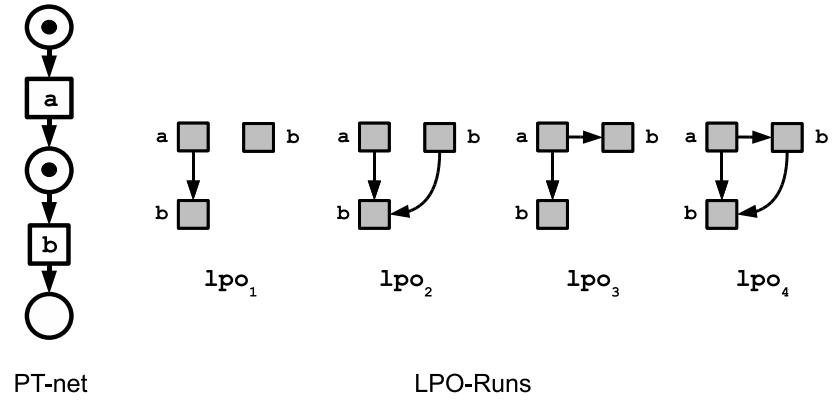


Fig. 5. A PT-net with four of its LPO-runs. The LPOs lpo_2, lpo_3 and lpo_4 are step linearizations of lpo_1. The LPO-run lpo_1 is minimal. The LPO lpo_4 is linear.

This means, after the occurrence of each prefix of lpo there are enough tokens for the occurrence of the multiset of transitions occurrences directly following the prefix.

In figures we often omit transitive arrows of LPOs for a clearer presentation.

2.3 Regions of Finite Languages

The formal problem statement, which we consider from now, is:

Given: A prefix-closed and extension-closed finite language L of LPOs over a finite alphabet of transition names T.

Searched: A PT-net N with set of transitions T such that all LPOs in L are LPO-runs of N and N has a minimal number of additional LPO-runs.

As explained in the introduction, for the computation of places of N so-called regions are defined. In this subsection we define two different types of PT-net regions of finite languages of LPOs as non-negative integral solutions of appropriate linear systems of the form $\mathbf{A}_L \cdot \mathbf{x} \leq \mathbf{b}_L$. For these definitions and in examples we only consider those LPOs from L, which are not extensions or prefixes of other LPOs from L. If a place is feasible w.r.t. these LPOs, then this place is feasible w.r.t. L, since the set of LPO-runs of a PT-net is prefix- and extension-closed. Throughout the rest of this subsection we use the language shown in Figure 6 as a running example. It is enough to consider the LPOs lpo_1 and lpo_2, since the other LPOs are prefixes or extensions of lpo_1.

Transition-Regions. A *(PT-net) transition-region* $\mathbf{r}$ directly defines the parameters of a place $p_\mathbf{r}$ of PT-nets, i.e. it determines the numbers $m_0(p_\mathbf{r})$ and $W(p_\mathbf{r}, t)$ and $W(t, p_\mathbf{r})$ for each $t \in T$. If $T = \{t_1, \ldots, t_n\}$, then $\mathbf{r}$ is given as a $(2n + 1)$-tuple

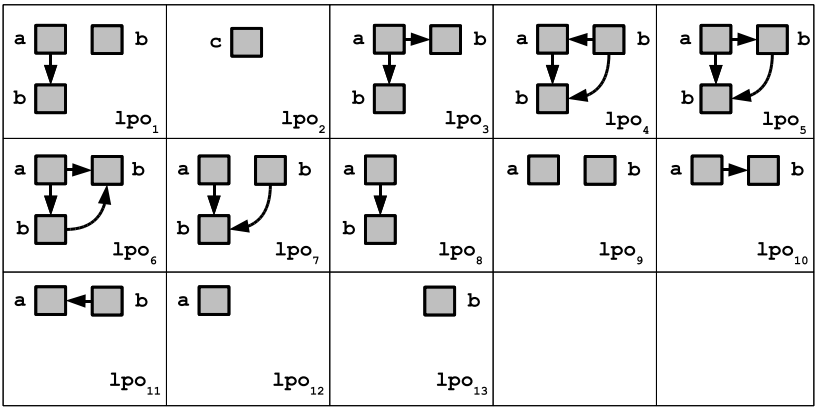


Fig. 6. Running example language

$\mathbf{r} = (r_0, \ldots, r_{2n})$ of non-negative integers. Its components define these numbers via $m_0(p_\mathbf{r}) = r_0$, $W(p_\mathbf{r}, t_i) = r_i$ and $W(t_i, p_\mathbf{r}) = r_{n+i}$ for $i \in \{1, \ldots, n\}$. In the running example, denote $t_1 = a$, $t_2 = b$ and $t_3 = c$.

Since a region $\mathbf{r}$ is intended to define a *feasible* place $p_\mathbf{r}$, it is required to satisfy a property $(f)_L$ ensuring that $p_\mathbf{r}$ is feasible w.r.t. L. Remember that $p_\mathbf{r}$ is feasible w.r.t. L if the net resulting from adding $p_\mathbf{r}$ still generates at least L. For this, the property $(f)_L$ formalizes that for each cut of events there are enough tokens in $p_\mathbf{r}$ for the occurrence of the corresponding step of transitions after the occurrence of the prefix preceeding the cut (which can be the empty prefix). For example, in the running example the transition step $(1a, 1b)$ must be able to occur after the empty prefix, i.e. in the initial marking (see Figure 7). This means, $p_\mathbf{r}$ has to satisfy $m_0(p_\mathbf{r}) \geq W(p_\mathbf{r}, a) + W(p_\mathbf{r}, b)$, i.e. $r_0 \geq r_1 + r_2$.

The definition of $(f)_L$ for a finite language L of LPOs and PT-nets is as follows: For each $lpo = (V, <, l) \in L$ and for each cut C of lpo we require

$$r_0 + \sum_{i=1}^{n} l(V')(t_i)(r_{n+i} - r_i) - \sum_{i=1}^{n} l(C)(t_i) r_i \geq 0,$$

where $V' = \{v \in V \mid v < C\}$. This is the case if and only if (for each $lpo \in L$ and for each cut C of lpo) $\mathbf{a}_{lpo,C} \cdot \mathbf{r} \leq 0$ for $\mathbf{a}_{lpo,C} = (a_{C,0}, \ldots, a_{C,2n})$ defined by:

$$\mathbf{a}_{C,j} = \begin{cases} -1 & \text{if } j = 0, \\ l(V' \cup C)(t_j) & \text{if } j \in \{1, \ldots, n\}, \\ -l(V')(t_{j-n}) & \text{if } j \in \{n+1, \ldots, 2n\}. \end{cases}$$

For the cut C corrsponding to the transition step $(1a, 1b)$ in the running example we require $r_0 - r_1 - r_2 \geq 0$. This is the case if and only if $\mathbf{a}_{lpo1,C} \cdot \mathbf{r} \leq 0$ for

$$\mathbf{a}_{lpo1,C} = (-1, 1, 1, 0, 0, 0, 0).$$

Definition 8 (Transition-Region). *A tuple* $\mathbf{r}$ *as above is called a* transition-region *if it satisfies* $(f)_L$.

Theorem 1 ([27]). *A tuple* $\mathbf{r}$ *satisfies* $(f)_L$ *if and only if* $p_\mathbf{r}$ *is feasible w.r.t.* L.

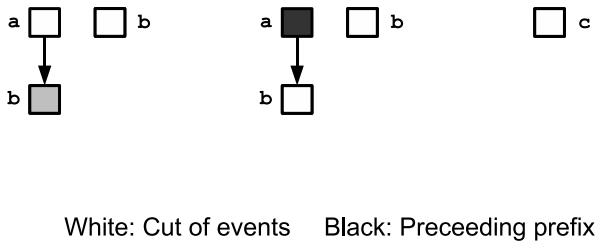


Fig. 7. All cuts of the LPOs lpo_1 and lpo_2 together with their preceeding prefixes

Let $\mathbf{A}_L$ be the matrix consisting of all rows $\mathbf{a}_{lpo,C}$ for LPOs $lpo \in L$ and cuts C of lpo. Since L is assumed to be finite, $\mathbf{A}_L$ is finite. Thus, the set of all regions can be computed as the set of all integral solutions of the homogenous linear inequation system $\mathbf{A}_L \cdot \mathbf{x} \leq \mathbf{0}$. In the running example, $\mathbf{A}_L$ looks as follows (compare Figure 7):

$$\begin{pmatrix} -1 & 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 2 & 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Solutions are for example $\mathbf{r} = (1,1,0,1,0,0,0)$ with corresponding place p_1, $\mathbf{r} = (1,0,1,1,1,0,0)$ with corresponding place p_2 and $\mathbf{r} = (0,0,0,0,0,1,0)$ with corresponding place p_3 in Figure 8.

Theorem 2 ([27]). *If L is finite then there is a finite matrix $\mathbf{A}_L$ such that the set of transition-regions is the set of solutions of the linear inequation system $\mathbf{A}_L \cdot \mathbf{x} \leq \mathbf{0}$.*

Token Flow Regions. A *token flow-region* $\mathbf{r}$ defines a place $p_{\mathbf{r}}$ indirectly by determining the *token flow* w.r.t. this place between transition occurrences in LPOs from L, i.e. by directly determining the number of tokens produced by a transition occurrence which are consumed by a subsequent transition occurrence in an LPO specified in L.

Such numbers are assigned to the arrows between transition occurrences of LPOs. Moreover, for each transition occurrence the number of tokens consumed from the initial marking and the number of tokens which are produced but not further consumed by other transition occurrences are considered. Finally, there may be tokens in the initial marking which are not consumed by any transition occurrence of an LPO.

If $W = \bigcup_{(V,<,l)\in L} V$ is the set of nodes of LPOs in L and $E = \bigcup_{(V,<,l)\in L} <$ is the set of arrows of LPOs in L, then a *(PT-net) token flow-region* $\mathbf{r}$ is given as a tuple $\mathbf{r} = (r_i)_{i\in W\times\{in,out\}\cup E\cup L}$ of non-negative integers. Its components define

- the number of tokens an event $v \in W$ consumes from the initial marking by $r_{v,in}$,
- the number of tokens produced by an event v and not consumed by a subsequent event by $r_{v,out}$,

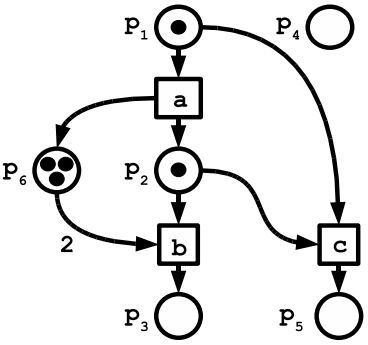


Fig. 8. Some solution places for the example language

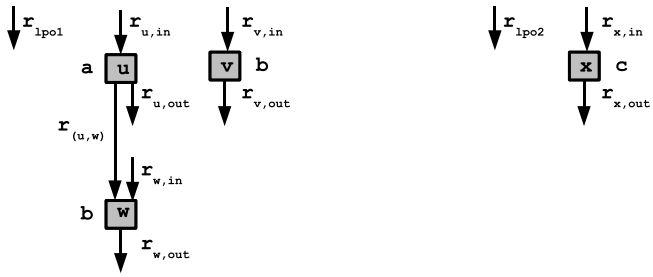


Fig. 9. Illustration of token flow regions for the running example

- the number of tokens produced by an event v and consumed by an event w for $e = (v, w) \in E$ by r_e,
- the number of tokens in the initial marking, which are not consumed by any subsequent event of an LPO $lpo \in L$ by r_{lpo}.

Figure 9 illustrates all these numbers for the running example. For example, $r_{u,in}$ represents the number of tokens, this occurrence of transition a consumes from the initial marking of the represented place. The number $r_{(u,w)}$ represented the number of tokens, which are produced by transition a in the represented place and then consumed by the following transition occurrence of b. The number $r_{u,out}$ represents the number of tokens, this occurrence of transition a produces in the the represented place and which are not consumed by further transitions. For the running example, we denote

$$\mathbf{r} = (r_{lpo1}, r_{lpo2}, r_{u,in}, r_{v,in}, r_{w,in}, r_{x,in}, r_{u,out}, r_{v,out}, r_{w,out}, r_{x,out}, r_{(u,w)}).$$

A token-flow region $\mathbf{r}$ defines a PT-net-place $p_{\mathbf{r}}$ as follows:

- $m_0(p_{\mathbf{r}}) = r_{lpo} + \sum_{v \in V} r_{v,in}$ for some LPO $lpo = (V, <, l) \in L$ – the sum is called *initial token flow* of lpo. The initial token flow of lpo_1 equals $r_{lpo1} + r_{u,in} + r_{v,in} + r_{w,in}$.
- $W(p_{\mathbf{r}}, t) = r_{v,in} + \sum_{e=(u,v) \in E} r_e$ for some LPO $lpo = (V, <, l) \in L$ and $v \in V$ with $l(v) = t$ – the sum is called *intoken flow* of v. The intoken flow event w is $r_{(u,w)} + r_{w,in}$.
- $W(t, p_{\mathbf{r}}) = r_{v,out} + \sum_{e=(v,u) \in E} r_e$ for some LPO $lpo = (V, <, l) \in L$ and $v \in V$ with $l(v) = t$ – the sum is called *outtoken flow* of v. The outtoken flow event u computes $r_{(u,w)} + r_{u,out}$.

This construction is still dependent on the choice of $lpo = (V, <, l) \in L$ and $v \in V$, thus $p_{\mathbf{r}}$ is not uniquely defined. Therefore, we require $\mathbf{r}$ to fulfill a property $(wd)_L$ which makes $p_{\mathbf{r}}$ defined independently from the choice of $lpo = (V, <, l) \in L$ and $v \in V$. The property $(wd)_L$ states the following:

– *The initial token flows of different LPOs are equal*:

$$r_{lpo} + \sum_{v \in V} r_{v,in} = r_{lpo'} + \sum_{v' \in V'} r_{v',in}$$

for LPOs $lpo = (V, <, l), lpo' = (V', <', l')$ from L. This property can be expressed as a linear equation system $\mathbf{A}_{L,a} \cdot \mathbf{r} = \mathbf{0}$ as follows:
Let $L = \{lpo_1, lpo_2, \ldots, lpo_n\}$ and $lpo_k = (V_k, <_k, l_k)$. Then, for each $k \geq 2$, the matrix $\mathbf{A}_{L,a}$ has a row $\mathbf{a}_k = (a_{k,i})_{i \in W \times \{in,out\} \cup E \cup L}$ defined by

$$\mathbf{a}_{k,i} = \begin{cases} 1 & \text{if } i = (v, in) \text{ for } v \in V_k, \\ 1 & \text{if } i = lpo_k, \\ -1 & \text{if } i = (v', in) \text{ for } v' \in V_{k-1} \\ -1 & \text{if } i = lpo_{k-1}, \\ 0 & \text{else.} \end{cases}$$

For example, the two LPOs of the running example shown in Figure 9 should have the same initial token flow, i.e. we require

$$r_{lpo1} + r_{u,in} + r_{v,in} + r_{w,in} = r_{lpo2} + r_{x,in}.$$

This is satisfied if and only if $\mathbf{a}_2 \cdot \mathbf{r} = \mathbf{0}$ for

$$\mathbf{a}_2 = (-1, 1, -1, -1, -1, 1, 0, 0, 0, 0, 0).$$

– *The intoken flows of equally labelled events are equal*:

$$r_{v,in} + \sum_{e=(u,v) \in <} r_e = r_{v',in} + \sum_{e=(u,v') \in <'} r_e$$

for $v \in V$, $v' \in V'$, $(V, <, l), (V', <', l') \in L$ and $l(v) = l'(v')$. This property can be expressed as a linear equation system $\mathbf{A}_{L,b} \cdot \mathbf{r} = \mathbf{0}$ as follows: Let $W_t = \{v \in W \mid l(v) = t\} = \{v_1^t, v_2^t, \ldots, v_n^t\}$ be the set of all t-labeled events for $t \in T$. Then, for each t and each $k \geq 2$, the matrix $\mathbf{A}_{L,b}$ has a row $\mathbf{b}_k^t = (b_{k,i}^t)_{i \in W \times \{in,out\} \cup E \cup L}$ defined by

$$\mathbf{b}_{k,i}^t = \begin{cases} 1 & \text{if } i = (v_k^t, in) \vee i = (u, v_k^t), \\ -1 & \text{if } i = (v_{k-1}^t, in) \vee i = (u, v_{k-1}^t) \\ 0 & \text{else.} \end{cases}$$

For example, the two occurrences v and w of transition b shown in Figure 9 should have the same intoken flow, i.e. we require

$$r_{v,in} = r_{w,in} + r_{(u,w)}.$$

If we denote $v_1^b = v$ and $v_2^b = w$, this is satisfied if and only if $\mathbf{b}_2^b \cdot \mathbf{r} = \mathbf{0}$ for

$$\mathbf{b}_2^b = (0, 0, 0, -1, 1, 0, 0, 0, 0, 0, 1).$$

– *The outtoken flows of equally labeled events are equal*:

$$r_{v,out} + \sum_{e=(v,u)\in <} r_e = r_{v',out} + \sum_{e=(v',u)\in <'} r_e$$

for $v \in V$, $v' \in V'$, $(V, <, l), (V', <', l') \in L$ and $l(v) = l'(v')$. This property can be expressed as a linear equation system $\mathbf{A}_{L,c} \cdot \mathbf{r} = \mathbf{0}$ as follows: For each t and each $k \geq 2$ the matrix $\mathbf{A}_{L,c}$ has a row $\mathbf{c}^t_k = (c^t_{k,i})_{i\in W\times\{in,out\}\cup E\cup L}$ defined by

$$\mathbf{c}^t_{k,i} = \begin{cases} 1 & \text{if } i = (v^t_k, out) \vee i = (v^t_k, u), \\ -1 & \text{if } i = (v^t_{k-1}, out) \vee i = (v^t_{k-1}, u) \\ 0 & \text{else.} \end{cases}$$

For example, the two occurrences v and w of transition b shown in Figure 9 should have the same outtoken flow, i.e. we require

$$r_{v,out} = r_{w,out}.$$

If we denote $v^b_1 = v$ and $v^b_2 = w$, this is satisfied if and only if $\mathbf{c}^b_2 \cdot \mathbf{r} = \mathbf{0}$ for

$$\mathbf{c}^b_2 = (0, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0).$$

It can be shown that the place $p_{\mathbf{r}}$ is feasible w.r.t. L by construction (it is possible to construct a process of the synthesized net directly from token flows).

Definition 9 (Token Flow-Region). *A tuple* $\mathbf{r}$ *as above is called a* token flow-region *if it satisfies* $(wd)_L$.

For the property $(wd)_L$ the following theorem holds for PT-net places [5]:

Theorem 3. *A tuple* $\mathbf{r}$ *satisfies* $(wd)_L$ *if and only if* $p_{\mathbf{r}}$ *is feasible w.r.t.* L.

Let $\mathbf{A}_L$ be the matrix consisting of all rows from the matrices $\mathbf{A}_{L,a}$, $\mathbf{A}_{L,b}$ and $\mathbf{A}_{L,c}$. Since L is assumed to be finite, $\mathbf{A}_L$ is finite. Thus, the set of all token flow-regions can be computed as the set of all integral solutions of the homogenous linear equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$. If we denote $v^b_1 = v$ and $v^b_2 = w$, $\mathbf{A}_L$ looks as follows for the running example (the matrices $\mathbf{A}_{L,a}$, $\mathbf{A}_{L,b}$ and $\mathbf{A}_{L,c}$ each consist exactly of the one row already shown):

$$\begin{pmatrix} -1 & 1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \end{pmatrix}$$

Figure 10 shows a solution. This solution represents the place p_2 from Figure 8.

Theorem 4 ([5,27]). *If* L *is finite then there is a finite matrix* $\mathbf{A}_L$ *such that the set of token flow-regions is the set of solutions of the linear equation system* $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$.

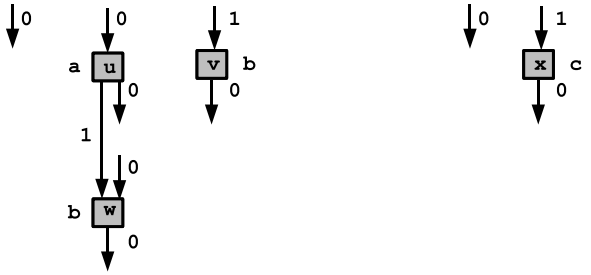


Fig. 10. Illustration of the token flow region $\mathbf{r} = (0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1)$

2.4 Finite Representations

As shown in the last paragraph, the set of regions can be computed as the set of non-negative integer solutions of a homogenuous equation ot inequation system. Such systems have an infinite set of solutions. In this subsection we present two possibilities to finitely represent this set of solutions.

Separating Representation. One idea to derive a finite representation is to separate behavior specified in L from behavior not specified in L by a finite set of regions. For this, one defines a finite set of executions L^c with $L \cap L^c = \emptyset$ satisfying that $L(N) \cap L^c = \emptyset \implies L(N) = L$ for each net N. Then for each $w \in L^c$ one tries to find a region $\mathbf{r}(w)$ such that w is not an execution of the net having the place $p_{\mathbf{r}(w)}$, i.e. a region which *separates L from w*. The elements of L^c are called *wrong continuations*. If such a region exists, then the corresponding place is added to the net N_{sep} called *separating representation* of N_{sat}.

There is an exact solution of the synthesis problem if and only if for each $w \in L^c$ there is such a region $\mathbf{r}(w)$. In case L is a net language (of the considered net class), it holds $L(N_{sep}) = L(N_{sat}) = L$, i.e N_{sep} is a possible solution.

If L is not a net language, L^c does not have in general the property $L(N_{sat}) = L(N_{sep})$. One common approach is to define a wrong continuation w as an LPO extending an LPO of L by one event. If there is no place separating L from such a wrong continuation w, this does not mean that there are no places separating L from further "continuations" of w. In order to achieve $L(N_{sat}) = L(N_{sep})$, also all continuations of wrong continuations which cannot be separated from L must be considered. In general, there is no *finite* set L^c with $L \cap L^c = \emptyset$ satisfying $L(N_{sat}) = L(N_{sep})$ [10]. Thus, the separating representation is not necessarily the best approximation to a solution of the synthesis problem generating L.

In the following we construct a finite set L^c. Remember that $lpo \in L$ is a run of a net N if and only if each step linearization of lpo is a step execution of N. Denote by L^{step} the set of step linearizations of LPOs in L. In order to define wrong continuations we extend elements from L^{step} by one event as follows:

Definition 10 (Wrong Continuation of LPO-languages). *Let $\sigma = \alpha_1 \dots \alpha_{n-1}\alpha_n \in L^{step}$ and $t \in T$ such that $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n + t) \notin L^{step}$, where α_n is allowed to be the empty step. Then $w_{\sigma,t}$ is called* wrong continuation of L.

We call $\alpha_1 \dots \alpha_{n-1}$ the prefix *and $\alpha_n + t$ the* follower step of the wrong continuation.

The following table lists L^{step} together all wrong continuations of the running example (multisets are denoted as sums of singleton multisets):

$(a+b)b\epsilon$	$(a+b)ba$ $(a+b)bb$ $(a+b)bc$	$abb\epsilon$	$abba$ $abbb$ $abbc$	$bab\epsilon$	$baba$ $babb$ $babc$
$a(2b)\epsilon$	$a(2b)a$ $a(2b)b$ $a(2b)c$	$(a+b)\epsilon$	$(a+b)a$ $(a+b)c$	$ab\epsilon$	aba abc
$ba\epsilon$	baa bac	$(a+b)b$	$(a+b)(b+a)$ $(a+b)(2b)$ $(a+b)(b+c)$	$abb\epsilon$	$ab(b+a)$ $ab(2b)$ $ab(b+c)$
bab	$ba(b+a)$ $ba(2b)$ $ba(b+c)$	$a\epsilon$	aa ac	ab	$a(b+a)$ $a(b+c)$

$a(2b)$	$a(2b+a)$ $a(3b)$ $a(2b+c)$	$b\epsilon$	bb bc	ba	$b(2a)$ $b(a+b)$ $b(a+c)$
$c\epsilon$	ca cb cc	$\epsilon\epsilon$		ϵa	$\epsilon(2a)$ $\epsilon(a+c)$
ϵb	$\epsilon(2b)$ $\epsilon(b+c)$	ϵc	$\epsilon(c+a)$ $\epsilon(c+b)$ $\epsilon(2c)$	$\epsilon(a+b)$	$\epsilon(2a+b)$ $\epsilon(a+2b)$ $\epsilon(a+b+c)$

To prohibit a wrong continuation, one needs to find a feasible place p such that after occurrence of its prefix there are not enough tokens in p to fire its follower step. A a prefix of wrong continuations corresponds to a prefix $(V', <, l)$ of an LPO $lpo = (V, <, l) \in L$, which is stepwise linearized by $\alpha_1 \dots \alpha_{n-1}$. A follower step of such a prefix can be constructed by taking a subset S of its direct successors $\{v \in V \setminus V' \mid u < v \implies u \in V'\}$ and add a labelled event z parallel to this subset. That means, wrong continuations can be represented on the level of LPOs, where wrong continuations having the same follower step and whose prefixes stepwise linearize the same LPO-prefix need not be distinguished. Figure 11 shows some representations of wrong continuations of the running example.

Since the follower marking after the occurrence of $(V', <, l)$ only depends on the number of occurrences of each transition in V', but not on their ordering, it is enough to represent $(V', <, l)$ by the multiset $l(V')$. Altogether, the set of all wrong continuations

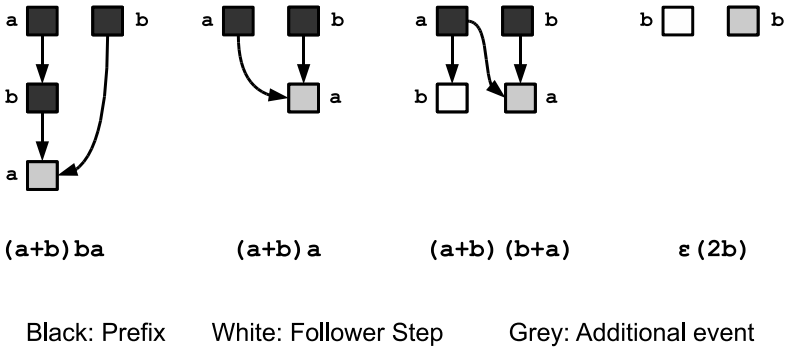


Fig. 11. Some wrong continuations on the level of LPOs

can be constructed as the set of all pairs of multisets $(l(V'), l(S \cup \{z\}))$, where $(V', <, l)$ is a prefix of some LPO in L, S is a subset of direct successors of $(V', <, l)$ and z is an additional labelled event. For example, the wrong continuations shown in Figure 11 are represented in the form $(a + 2b, a), (a + b, a), (a + b, a + b), (\epsilon, 2c)$ (from left to right).

With these notations and the notations from the previous subsection we directly deduce the following statements for transition regions and token flow regions.

If $\mathbf{r}$ is a transition region, $T = \{t_1, \ldots, t_n\}$, $C = S \cup \{z\}$ and $l(z) = t$ then $w_{\sigma,t}$ is not a step execution w.r.t. $p_{\mathbf{r}}$ if and only if

$$r_0 + \sum_{i=1}^{n} l(V')(t_i)(r_{n+i} - r_i) - \sum_{i=1}^{n} l(C)(t_i) r_i < 0.$$

This is the case if and only if $\mathbf{d}(w_{\sigma,t}) \cdot \mathbf{r} < 0$ for $\mathbf{d}(w_{\sigma,t}) = (d_0, \ldots, d_{2n})$ defined by:

$$d_j = \begin{cases} 1 & \text{if } j = 0, \\ -l(V' \cup C)(t_j) & \text{if } j \in \{1, \ldots, n\}, \\ l(V')(t_{j-n}) & \text{if } j \in \{n+1, \ldots, 2n\}. \end{cases}$$

For example, for the left most wrong continuation in Figure 11 we require $r_0 + ((r_4 - r_1) + 2(r_5 - r_2)) - (r_1) < 0$ (remember $t_1 = a$, $t_2 = b$ and $t_3 = c$). This is the case if and only if $\mathbf{d}(w_{\sigma,t}) \cdot \mathbf{r} < 0$ for

$$\mathbf{d}(w_{\sigma,t}) = (1, -2, -2, 0, 1, 2, 0).$$

The region

$$\mathbf{r} = (1, 1, 0, 1, 0, 0, 0)$$

(corresponding to place p_1 in Figure 8) is a solution which prohibits this wrong continuation.

If $\mathbf{r}$ is a token flow region, then the number of tokens in the place $p_{\mathbf{r}}$ after the occurrence of a prefix $(V', <, l)$ of some LPO $lpo = (V, <, l))$ equals the initial token flow of lpo minus the sum of intoken flows of events in V' plus the sum of outtoken flows of

events in V'. This sum needs to be smaller than the sum of intoken flows of events in C. Formally, if v_t an arbitrary event with label t then $w_{\sigma,t}$ is not a step execution w.r.t. $p_{\mathbf{r}}$ if and only if

$$r_{lpo} + \sum_{u \in V \setminus (V' \cup S)} r_{u,in} + \sum_{v \in V'} r_{v,out} + \sum_{v \in V',\, u \in V \setminus (V' \cup S)} r_{v,u} - (r_{(v_t,in)} + \sum_{u < v_t} r_{(u,v_t)}) < 0.$$

This is the case if and only if $\mathbf{d}(w_{\sigma,t}) \cdot \mathbf{r} < 0$ for $\mathbf{d}(w_{ext(t)}) = (d_i)_{i \in W \times \{in,out\} \cup E \cup L}$ defined by:

$$d_i = \begin{cases} 1 & \text{if } i = lpo, \\ 1 & \text{if } i = (u,in) \wedge u \in V \setminus (V' \cup S) \wedge u \neq v_t, \\ 1 & \text{if } i = (v,out) \wedge v \in V', \\ 1 & \text{if } i = (v,u) \wedge v \in V' \wedge u \in V \setminus (V' \cup S) \wedge u \neq v_t, \\ -1 & \text{if } i = (v_t,in) \wedge v_t \notin V \setminus (V' \cup S), \\ -1 & \text{if } i = (u,v_t) \wedge v_t \notin V \setminus (V' \cup S), \\ 0 & \text{else.} \end{cases}$$

For example, for the right most wrong continuation in Figure 11 we require $r_{lpo1} + r_{u,in} + r_{w,in} - r_{v,in} < 0$ (we use the notations from Figure 9 and $V' = \emptyset$ and $S = \{v\}$) This is the case if and only if $\mathbf{d}(w_{\sigma,t}) \cdot \mathbf{r} < 0$ for

$$\mathbf{d}(w_{\sigma,t}) = (1, 0, 1, -1, 1, 0, 0, 0, 0, 0, 0).$$

The region

$$\mathbf{r} = (0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1)$$

(illustrated in Figure 10 and corresponding to place p_2 in Figure 8) is a solution which prohibits this wrong continuation.

Summarizing, a region separating L from some wrong continuation w is computed as a solution of an adequate homogenuous linear inequation system. Such a system consists of the equations/inequations given by $\mathbf{A}_L$ defining regions and an additional row $\mathbf{d}(w)$ which is defined in such a way that $\mathbf{d}(w) \cdot \mathbf{r} < 0$ if and only if w is not an execution of the net having the place $p_{\mathbf{r}}$. There are effective algorithms to compute a non-negative integer solution of the resulting system. One example is the simplex algorithm, which allows to compute a solution of such a system which addionally minimizes or maximizes a given linear target function. Such a target function may be used to compute simple places. This is of high importance in practise where the derived system model should be as clear and compact as possible. For example, the places p_2 and p_6 in Figure 8 both prohibit the most right wrong continuation from Figure 11, but place p_2 is much more simple and intuitive.

Definition 11 (Linear Target Function of LPO-languages). *A* linear target function (of an LPO-language) *is a function of the form* $T(\mathbf{r}) = \mathbf{m} \cdot \mathbf{r}$ *for regions* $\mathbf{r}$*, where* $\mathbf{m}$ *is the vector defining* T.

It is possbile to define $\mathbf{m}$ in such a way that the sum of initial marking and arc weight on incoming and outgoing edges w.r.t. $p_{\mathbf{r}}$ is minimized when T is minimized.

If $\mathbf{r}$ is a transition region and $T = \{t_1, \ldots, t_n\}$, then $\mathbf{m}$ is defined through

$$\mathbf{m} = (1, \ldots, 1),$$

because $\mathbf{m} \cdot \mathbf{r} = \sum_{i=0}^{2n} m_i r_i = m_0(p_{\mathbf{r}}) + \sum_{i=1}^{n} (W(p_{\mathbf{r}}, t_i) + W(t_i, p_{\mathbf{r}}))$.

If $\mathbf{r}$ is a token flow region, then the initial marking of $p_{\mathbf{r}}$ can be computed as the initial token flow of some LPO in L and the arc weights on incoming and outgoing edges w.r.t. $p_{\mathbf{r}}$ can be computed as intoken and outtoken flows of some events $v_1, \ldots, v_n$ satisfying $l(v_j) = t_j$ (we omit a formal definition here). In the running example $\mathbf{m}$ can be defined through

$$\mathbf{m} = (0, 1, 1, 1, 0, 2, 1, 1, 0, 1, 1),$$

because $m_0(p_{\mathbf{r}}) = r_{lpo2} + r_{x,in}$, $W(p_{\mathbf{r}}, a) = r_{u,in}$, $W(p_{\mathbf{r}}, b) = r_{v,in}$, $W(p_{\mathbf{r}}, c) = r_{x,in}$, $W(a, p_{\mathbf{r}}) = r_{u,out} + r_{u,w}$, $W(b, p_{\mathbf{r}}) = r_{v,out}$ and $W(c, p_{\mathbf{r}}) = r_{x,out}$.

Definition 12 (Minimal Places). *Given a target function T, a feasible place is called* minimal w.r.t. a wrong continuation*, if it minimizes T among all feasible places prohibiting the wrong continuation.*

Figure 12 illustrates that a wrong continuation may be prohibited by several different feasible places. The target function T has different values for these places. The smaller the value of T is, the more simple is the place.

On the other side, Figure 12 shows that a feasible place may prohibit several wrong continuations. Sometimes, a place prohibits all wrong continuations another place prohibits. Such places are more expressive in the sense that, that less places of this kind are needed to prohibit all wrong continuations.

Definition 13 (Expressive Places). *A feasible place p is called* more expressive than *second feasible place p', if the set of wrong continuations prohibited by p' is contained in the set of wrong continuations prohibited by p.*

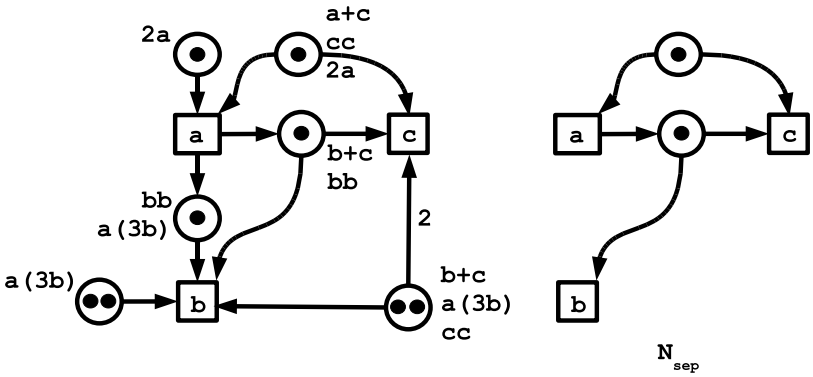


Fig. 12. Left side: Some feasible places together with some of the wrong continuations they prohibit. Right side: A solution

A place which is more expressive than another place is not minimal w.r.t. some wrong continuations. This means, it is necessary to find a trade-off between simple (minimal) and expressive places.

Since feasible places in general prohibit more than one wrong contintuation, N_{sep} is constructed as follows:

1. Compute a sequence of all wrong continuations.
2. For each wrong continuation w (considered in the given order):
   a. If w is prohibited by a previously computed place, skip w.
   b. If w is not prohibited by a previously computed place, then compute and add a place prohibiting w (if possible.

The above considerations imply that it depends on the considered order of wrong continuations, which places are computed. On the right side of Figure 12 a possible solution is shown. It is advantageous to choose an order such that more expressive places are computed first. There are several methods for constructing an appropriate order of wrong continuations.

In general, the number of wrong continuations is exponential in the number of nodes in L (first step of the algorithm). Since feasible places often prohibit more than one wrong continuation, the number of computed places is usually much smaller. For step 2a. we need to test for every previously computed place, whether it solves the inequation system. The simplex algorithm for solving step 2b. needs worst case exponential time (there are other worts case polynomial algorithms, but probabilistic and experimental results show that the Simplex algorithm has a significantly faster average runtime).

Basis Representation. For systems of the form $\mathbf{A}_L \cdot \mathbf{r} = 0$ or $\mathbf{A}_L \cdot \mathbf{r} \leq 0$ there is a so called *basis representation* of the set of all non-negative solutions. This means there are non-negative *basis-solutions* $\mathbf{y}_1, \ldots, \mathbf{y}_n$ such that each solution $\mathbf{x}$ is a non-negative linear combination of $\mathbf{y}_1, \ldots, \mathbf{y}_n$, i.e.

$$\mathbf{x} = \sum_{i=1}^{n} \lambda_i \mathbf{y}_i$$

for real numbers $\lambda_1, \ldots, \lambda_n \geq 0$. In the case that all values in $\mathbf{A}_L$ are integral (this is the case here) also the values of $\mathbf{y}_1, \ldots, \mathbf{y}_n$ can be chosen integral. If p_i is the place defined by $\mathbf{y}_i$ and N_{basis} is the net containing exactly the places $p_1, \ldots, p_n$, then $L(N_{basis}) = L(N_{sat})$ [5]. This means N_{basis} is also the best approximation to a solution of the synthesis problem generating L but N_{basis} is moreover finite. N_{basis} is called *basis representation* of N_{sat}.

In the worst case n can be exponential in the number of rows of $\mathbf{A}_L$, but in practice it is often small. There are effective algorithms to compute *minimal* basis solutions $\mathbf{y}_1, \ldots, \mathbf{y}_n$, where a solution $\mathbf{y} = (y_1, \ldots, y_k)$ is minimal, if there is no other solution $\mathbf{z} = (z_1, \ldots, z_k)$ satisfying $(\forall i : \ z_i \leq y_i) \wedge (\exists j : \ z_j < y_j)$. Figure 13 shows N_{basis} for the running example language. Observe that there are basis places which do not restrict the behavior of the net (filled with grey color). They are special cases of so called implicit places.

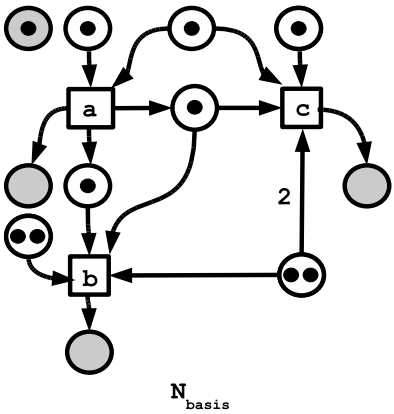


Fig. 13. The basis representation of the running example

Definition 14 (Implicit Place). *A place of a PT-net is called* implicit*, it the modified net without this place has the same set of executions.*

The are polynomial methods to detect some of the implicit places of a net which can be applied in a postprocessing phase to the synthesized net. An easy example are places whiich are dominated by one another place (w.r.t. the behavioral restriction).

Definition 15 (Dominating Place). *A place* p' dominates *another place* p*, if for some* $\lambda > 0$*:*

- $\lambda m_0(p) \geq \lambda m_0(p')$*,*
- $\lambda W(t,p) \geq \lambda W(t,p')$ *and* $\lambda W(p,t) \leq \lambda W(p',t)$ *for all transitions* t*.*

Dependent on the initial marking and the arc weights, a maximal number for λ can be determined. Then places can be compared pairwise. Advanced methods are able to detect places which are dominated by positive linear combinations of other places. Such places are implicit, too.

Since from the construction it is not clear whether N_{basis} is an exact solution of the synthesis problem, it is finally necessary to test whether $L(N_{basis}) = L$ or not. One possibility of such an equality test is to test whether no $w \in L^c$ (L^c was defined in the last paragraph) is an execution of N_{basis}. There are polynomial algoritms testing whether an LPO is an execution of a PT-net [25], but L^c in general contains exponential many LPOs in the number of nodes in L.

Discussion. Experiments in the first phase of the project SYNOPS showed that the so called separation representation produces Petri nets which are simpler and more compact, especially having less places [5]. It turned out that in the presence of much concurrency the use of the basis representation of token flow regions is most efficient (including an equality test), whereas in the case of low concurrency and much nondeterminism the use of the separation representation of transition regions ist advantageous (concerning runtime).

2.5 Regions of Infinite Languages

In this subsection we generalize the presented framework to inifinite LPO-languages. In order to finitely represent such languages we introduce a term based representation extending regular expressions. A term is built from a given finite set of LPOs through operators for iteration, parallel composition, sequential composition and union. The parallel composition operation represents concurrent LPOs. By the iteration operation infinite sets of LPOs can be constructed. For a term α we denote by $L(\alpha)$ the LPO-language represented by α.

The formal problem statement, which we consider from now, is:

Given: A term α over a finite alphabet of transition names T.
Searched: A PT-net N with set of transitions T such that all LPOs in $L(\alpha)$ are LPO-runs of N and N has a minimal number of additional LPO-runs.

As in the finite case, we will define transition regions and token flow regions of $L(\alpha)$ as non-negative integral solutions of appropriate linear systems of the form $\mathbf{A}_\alpha \cdot \mathbf{x} \leq \mathbf{b}_\alpha$. Throughout the rest of this subsection we use the language shown in Figure 14 as a running example, where zero, one and two iterations of the action b are shown and prefixes and extensions are omitted.

LPO-terms. Let $\mathcal{A}$ be a finite set of LPOs. For $A \in \mathcal{A}$ we write $A = (V_A, <_A, l_A)$. We denote by $\lambda = (\emptyset, \emptyset, \emptyset)$ the empty LPO. LPOs consisting only of one single event we denote by the label of this event.

Definition 16 (LPO-term). *The set of* LPO-terms *over a finite set of LPOs $\mathcal{A}$ is inductively defined as follows:*

- *The elements $A \in \mathcal{A}$ and λ are LPO-terms.*
- *Let α_1 and α_2 be LPO-terms. Then*
  - $\alpha_1; \alpha_2$ *(sequential composition),*
  - $\alpha_1 + \alpha_2$ *(union),*
  - $(\alpha_1)^*$ *(iteration),*
  - $\alpha_1 \parallel \alpha_2$ *(parallel composition)*

  are LPO-terms.

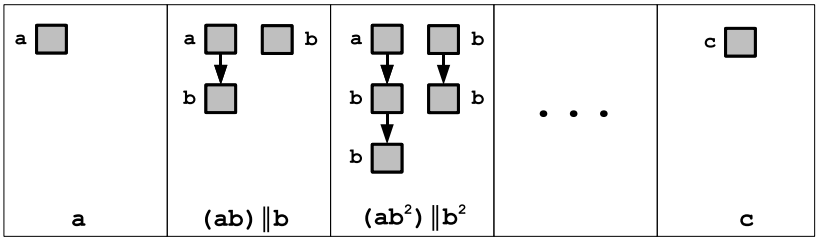


Fig. 14. Infinite example language, represented by a term

In the running example shown in Figure 14 $\mathcal{A} = \{a, b, c\}$ consists of three single event LPOs. If each LPO in $\mathcal{A}$ is single event LPO, an LPO-term defines a so called series rational sp-language [22,23].

We assign to an arbitrary LPO-term α a possibly infinite prefix and extension closed LPO-language $L(\alpha)$. The language $L(\alpha$ is defined as the prefix and extension closure of an appropriate LPO-language $K(\alpha)$. In order to construct $K(\alpha)$, we define the *sequential composition of LPOs* $A, B \in \mathcal{A}$ by

$$AB = (V_A \cup V_B, <_A \cup <_B \cup (V_A \times V_B), l_A \cup l_B),$$

the *parallel composition of LPOs* $A, B \in \mathcal{A}$ by

$$A \parallel B = (V_A \cup V_B, <_A \cup <_B, l_A \cup l_B),$$

and the *n-th iteration of an LPO* $A \in \mathcal{A}$ by

$$A^n = A^{n-1}A$$

for $n \in \mathbb{N}^+$ (we can assume that A, B have disjoint sets of nodes).

Definition 17 (LPO-language of an LPO-term). *We define inductively:*

- $K(\lambda) = \{\lambda\}$ *and* $K(A) = \{A\}$ *for* $A \in \mathcal{A}$,
- *Let* α_1 *and* α_2 *be LPO-terms. Then:*
  - $K(\alpha_1 + \alpha_2) = K(\alpha_1) \cup K(\alpha_2)$,
  - $K(\alpha_1; \alpha_2) = \{A_1A_2 \mid A_1 \in K(\alpha_1),\ A_2 \in K(\alpha_2)\}$,
  - $K((\alpha_1)^*) = \{A_1 \ldots A_n \mid A_1, \ldots, A_n \in K(\alpha_1)\} \cup \{\lambda\}$,
  - $K(\alpha_1 \parallel \alpha_2) = \{A_1 \parallel A_2 \mid A_1 \in K(\alpha_1),\ A_2 \in K(\alpha_2)\}$.

Some of the the LPOs in $K(\alpha)$ from the example LPO-term shown in Figure 14 are $c, a, ab, abb, \ldots, a \parallel b, a \parallel b^2, \ldots, (ab) \parallel b, (ab) \parallel b^2, \ldots$.

In the second part of this paper, LPO-languages which cannot be generated by LPO-terms are discussed. This means, LPO-languages generated by LPO-terms (over finite sets of LPOs) form a certain class of LPO-languages.

Regions of LPO-Terms. We now describe a technique to represent the infinite set $K(\alpha)$ by two finite sets of LPOs $R(\alpha)$ and $I(\alpha)$. Regions will be defined w.r.t. these finite sets.

An LPO A can occur arbitrarily often consecutively in a certain marking m if and only if it consumes in every place at most as many tokens as it produces in this place (then an occurrence of A does not reduce the number of tokens in this place). Consequently, if A can occur iterated in m, then another LPO B can occur after the occurrence of A^n for each $n \in \mathbb{N}$ if and only if it can occur in m, since an occurrence of A does not reduce the number of tokens in a place. This principle can be used to define the two finite sets $R(\alpha)$ and $I(\alpha)$.

For an LPO-term α the set $R(\alpha)$ is the set of, roughly speaking, all LPOs containing each iterated part of α at most once. The running example term $((a; b^*) \parallel b^*) + c$ contains to iterated b-labelled events, thus $R(((a; b^*) \parallel b^*) + c) = \{a, ab, a \parallel b, (ab) \parallel b, c\}$.

We call in $R(\alpha)$ the representation set. The first idea for the definition of regions is to ensure that the place defined by a region is feasible w.r.t. the representation set, i.e. we require that a transition region satisfies the property $(f)_{R(\alpha)}$ and a token flow regions satsfies the property $(wd)_{R(\alpha)}$ as defined for finite LPO-languages.

It remains to ensure that certain LPOs can occur iterated w.r.t. the place defined by a region. The set $I(\alpha)$ consist of, roughly speaking, all LPOs associated to iterated subterms of α. The running example term $((a; b^*) \parallel b^*) + c$ contains to iterated b-labelled events, thus $I(((a; b^*) \parallel b^*) + c) = \{b\}$. We call $I(\alpha)$ the iteration set. We will introduce an additional property $(i)_\alpha$ of regions which ensures that the LPOs in $I(\alpha)$ produce at least as many tokens as they consume in the place defined by a region. This implies that the place defined by a region is feasible w.r.t. $K(\alpha)$.

Definition 18 (Representation/Iteration set). *The* representation set $R(\alpha)$ *and the* iteration set $I(\alpha)$ *of an LPO-term* α *are defined inductively as follows* (α_1 *and* α_2 *are LPO-terms):*

$R(\lambda) = \{\lambda\}$	$I(\lambda) = \emptyset$
$R(A) = \{A\}$ *for* $A \in \mathcal{A}$	$I(A) = \emptyset$ *for* $A \in \mathcal{A}$
$R(\alpha_1 + \alpha_2) = R(\alpha_1) \cup R(\alpha_2)$	$I(\alpha_1 + \alpha_2) = I(\alpha_1) \cup I(\alpha_2)$
$R(\alpha_1; \alpha_2) = \{A_1 A_2 \mid A_1 \in R(\alpha_1),\ A_2 \in R(\alpha_2)\}$	$I(\alpha_1; \alpha_2) = I(\alpha_1) \cup I(\alpha_2)$
$R((\alpha_1)^*) = R(\alpha_1) \cup \{\lambda\}$	$I((\alpha_1)^*) = I(\alpha_1) \cup R(\alpha_1)$
$R(\alpha_1 \parallel \alpha_2) = \{A_1 \parallel A_2 \mid A_1 \in R(\alpha_1),\ A_2 \in R(\alpha_2)\}$	$I(\alpha_1 \parallel \alpha_2) = I(\alpha_1) \cup I(\alpha_2)$

This approach generalizes the ideas in [10] where the authors define regions by two finite sets representing a regular expression.

The requirement, that every LPO in $I(\alpha)$ produces at least as many tokens as it consumes in a place p, corresponds to the requirement, that the sum of tokens produced by all events of an LPO in $I(\alpha)$ exceeds the sum of tokens consumed by all events. For $lpo = (V, <, l)$ and some place p we define

$$Prod(lpo, p) := \sum_{t \in l(V)} l(V)(t)(W(t, p) - W(p, t)).$$

A region $\mathbf{r}$, i.e. property $(i)_\alpha$, is defined in such a way that $Prod(lpo, p_\mathbf{r}) \geq 0$ for each LPO $lpo \in I(\alpha)$. For the running example LPO-term we require $W(b, p_\mathbf{r}) - W(p_\mathbf{r}, b) \geq 0$.

The definition of $(i)_\alpha$ for transition regions of LPO-terms α and PT-nets is as follows, where a (PT-net) transition-region $\mathbf{r} = (r_0, \ldots, r_{2n})$ directly defines the parameters of a place $p_\mathbf{r}$ of PT-nets via $m_0(p_\mathbf{r}) = r_0$, $W(p_\mathbf{r}, t_i) = r_i$ and $W(t_i, p_\mathbf{r}) = r_{n+i}$, if $T = \{t_1, \ldots, t_n\}$ is the set of transition names occurring in LPOs from $K(\alpha)$. For each $lpo = (V, <, l) \in I(\alpha)$ we require

$$\sum_{i=1}^{n} l(V)(t_i)(r_{n+i} - r_i) \geq 0.$$

This is the case if and only if (for each $lpo \in I(\alpha)$) $\mathbf{i}_{lpo} \cdot \mathbf{r} \leq 0$ for $\mathbf{i}_{lpo} = (i_0, \ldots, i_{2n})$ defined by:

$$\mathbf{i}_j = \begin{cases} 0 & \text{if } j = 0, \\ l(V)(t_j) & \text{if } j \in \{1, \ldots, n\}, \\ -l(V)(t_{j-n}) & \text{if } j \in \{n+1, \ldots, 2n\}. \end{cases}$$

In the running example, denote $t_1 = a$, $t_2 = b$ and $t_3 = c$. For the LPO $b \in I(\alpha)$ in the running example we require $r_5 - r_2 \geq 0$. This is the case if and only if $\mathbf{i}_b \cdot \mathbf{r} \leq 0$ for

$$\mathbf{i}_b = (0, 0, 1, 0, 0, -1, 0).$$

Definition 19 (Transition-Region of LPO-term). *We call a tuple* $\mathbf{r}$ *a* transition-region *of an LPO-term* α *if it satisfies* $(f)_{R(\alpha)}$ *and* $(i)_\alpha$.

With these definitions and notions the following theorem holds:

Theorem 5. *A tuple* $\mathbf{r}$ *satisfies* $(f)_{R(\alpha)}$ *and* $(i)_\alpha$ *if and only if* $p_{\mathbf{r}}$ *is feasible w.r.t.* $K(\alpha)$.

Let $\mathbf{A}_\alpha$ be the matrix consisting of all rows of $\mathbf{A}_{R(\alpha)}$ and all rows $\mathbf{i}_{lpo}$ for LPOs $lpo \in I(\alpha)$. Then the set of all regions can be computed as the set of all integral solutions of the homogenous linear inequation system $\mathbf{A}_\alpha \cdot \mathbf{x} \leq \mathbf{0}$. In the running example, $\mathbf{A}_\alpha$ looks as follows (the set $R(\alpha)$ can be represented by the finite example language from the subsection on finite languages):

$$\begin{pmatrix} -1 & 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 2 & 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{pmatrix}$$

Solutions are for example $\mathbf{r} = (1, 1, 0, 1, 0, 0, 0)$ with corresponding place p_1, $\mathbf{r} = (1, 0, 1, 1, 1, 1, 0)$ and $\mathbf{r} = (0, 0, 0, 0, 0, 1, 0)$ with corresponding place p_3 of the PT-net shown in Figure 15.

Theorem 6. *If* α *is an LPO-term then there is a finite matrix* $\mathbf{A}_\alpha$ *such that the set of transition-regions is the set of solutions of the linear inequation system* $\mathbf{A}_\alpha \cdot \mathbf{x} \leq \mathbf{0}$.

The previous theorems are proven in [7] for token flow regions and easily carry over to transition regions.

If

$$W = \bigcup_{(V,<,l) \in R(\alpha)} V$$

is the set of nodes of LPOs in $R(\alpha)$ and

$$E = \bigcup_{(V,<,l) \in R(\alpha)} <$$

is the set of arrows of LPOs in $R(\alpha)$, then a *(PT-net) token flow-region* $\mathbf{r}$ *of an LPO-term* α is given as a tuple $\mathbf{r} = (r_i)_{i \in W \times \{in,out\} \cup E \cup L}$ of non-negative integers. Its components are interpreted and define places as for finite languages.

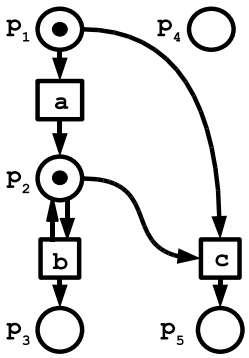


Fig. 15. Some solution places for the example LPO-term

The definition of $(i)_\alpha$ for token flow regions of LPO-terms α and PT-nets is as follows: For each $lpo \in I(\alpha)$ we require

$$\sum_{v \in V_{lpo}} (r_{v,out} + \sum_{e=(v,u)\in <} r_e - r_{v,in} - \sum_{e=(u,v)\in <} r_e) \geq 0,$$

where $(V_{lpo}, <, l)$ is a sub-LPO of some LPO $(V, <, l) \in R(\alpha)$ which is isomorphic to lpo. This is the case if and only if $\mathbf{i}_{lpo} \cdot \mathbf{r} \leq 0$ for $\mathbf{i}_{lpo} = (i_j)_{j \in W \times \{in,out\} \cup E \cup L}$ defined by:

$$\mathbf{i}_j = \begin{cases} 1 & \text{if } j = (v, in) \text{ for } v \in V_{lpo}, \\ 1 & \text{if } j = (u, v) \text{ for } v \in V_{lpo},\ u \in V \setminus V_{lpo}, \\ -1 & \text{if } i = (v, out) \text{ for } v \in V_{lpo}, \\ -1 & \text{if } j = (v, u) \text{ for } v \in V_{lpo},\ u \in V \setminus V_{lpo}, \\ 0 & \text{else.} \end{cases}$$

In the running example, it is enough to consider the LPOs $lpo1 = (ab) \parallel b$ and $lpo2 = c$ from $R(\alpha)$, since the other LPOs are prefixes of $lpo1$. Thus, $R(\alpha)$ can be treatened in the same way as the finite example language from the section on finite languages. As before, we denote

$$\mathbf{r} = (r_{lpo1}, r_{lpo2}, r_{u,in}, r_{v,in}, r_{w,in}, r_{x,in}, r_{u,out}, r_{v,out}, r_{w,out}, r_{x,out}, r_{(u,w)}),$$

where the events u, v, w and x are shown in Figure 9. If we represented the iterated LPO $b \in I((a; b^*) \parallel b^*)$ by the event v, then we require

$$r_{v,in} - r_{v,out} \leq 0.$$

This is satisfied if and only if $\mathbf{i}_b \cdot \mathbf{r} = \mathbf{0}$ for

$$\mathbf{i}_b = (0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0).$$

Definition 20 (Token Flow-Region of LPO-term). *A tuple* $\mathbf{r}$ *as above is called a* token flow-region *of an LPO-term* α *if it satisfies* $(wd)_{R(\alpha)}$ *and* $(i)_\alpha$.

With these definitions and notions the following theorem holds:

Theorem 7. *A tuple* $\mathbf{r}$ *satisfies* $(wd)_{R(\alpha)}$ *and* $(i)_\alpha$ *if and only if* $p_{\mathbf{r}}$ *is feasible w.r.t.* $K(\alpha)$.

Let $\mathbf{A}_L$ be the matrix consisting of all rows from the matrices $\mathbf{A}_{L,a}$, $\mathbf{A}_{L,b}$ and $\mathbf{A}_{L,c}$. Since L is assumed to be finite, $\mathbf{A}_L$ is finite. Thus, the set of all token flow-regions can be computed as the set of all integral solutions of the homogenous linear equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$. If we denote $v_1^b = v$ and $v_2^b = w$, $\mathbf{A}_L$ looks as follows for the running example (the matrices $\mathbf{A}_{L,a}$, $\mathbf{A}_{L,b}$ and $\mathbf{A}_{L,c}$ each consist exactly of the one row already shown):

Let $\mathbf{A}_{I(\alpha)}$ be the matrix consisting of all rows $\mathbf{i}_{lpo}$ for LPOs $lpo \in I(\alpha)$. The set of all regions can be computed as the set of all integral solutions of the homogenous linear inequation system $\mathbf{A}_{R(\alpha)} \cdot \mathbf{x} = \mathbf{0}$ and $\mathbf{A}_{I(\alpha)} \cdot \mathbf{x} \leq \mathbf{0}$. In the running example, the matrix of this system looks as follows:

$$\begin{pmatrix} -1 & 1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 16 shows a solution. This solution represents the place p_2 from Figure 15.

Theorem 8 ([7]). *If* α *is an LPO-term then there is a finite matrix* $\mathbf{A}_\alpha$ *such that the set of token flow-regions is the set of solutions of the linear inequation system* $\mathbf{A}_\alpha \cdot \mathbf{x} \leq \mathbf{0}$.

Finite Representation. As in the case of finite languages, the set of regions of an LPO-term is defined as the set of positive integral solutions of a homogenuous linear inequation system. Thus, it has a finite basis representation. Figure 17 shows the basis representation for the running example term.

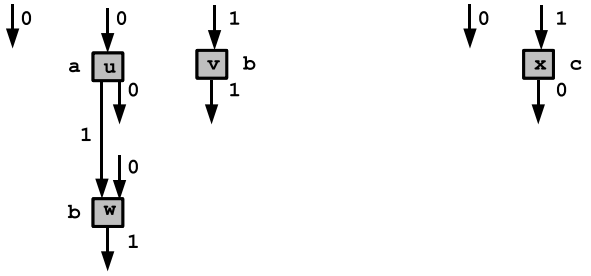


Fig. 16. Illustration of the token flow region $\mathbf{r} = (0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1)$

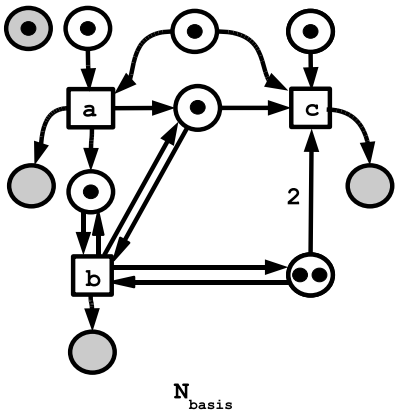


Fig. 17. The basis representation of the running example. The iterated transition b does not decrease the number of tokens in places

In order to compute a finite separation representation we need to finitely represent the infinite set of wrong continuations $K(\alpha)^c$. The next definition proposes such a finite representation:

Definition 21 (Wrong Continuation of LPO-term). *Let α be an LPO-term and let $\sigma = \beta_1 \dots \beta_{n-1}\beta_n \in R(\alpha)^{step}$ and $t \in T$ such that $w_{\sigma,t} = \beta_1 \dots \beta_{n-1}(\beta_n + t) \notin K(\alpha)^{step}$, where β_n is allowed to be the empty step. Then $w_{\sigma,t}$ is called* wrong continuation of α.

We call $\beta_1 \dots \beta_{n-1}$ the prefix *and $\beta_n + t$ the* follower step of the wrong continuation.

Note that this finite set of wrong continuations is usually a proper subset of $R(\alpha)^c$ because of the requirement $w_{\sigma,t} \notin K(\alpha)^{step}$. For the running example term $\alpha = ((a; b^*) \parallel b^*) + c$, $R(\alpha)^c$ is listed in the subsection on finite languages, where (for example) $bb \in R(\alpha)^c$ is not a wrong continuation of α.

For each wrong continuation w of α we search for a place $p_{\mathbf{r}}$ not only prohibiting w, but also an infinite set of wrong continuations $I(w) \subseteq K(\alpha)^c$. The idea is that, if the prefix of w contains the prefix of a sub-LPO (of some LPO in $R(\alpha)$) which can be iterated, then the follower step must be prohibited by $p_{\mathbf{r}}$ also after all finite iterations of this sub-LPO. In other words, $I(w)$ contains all wrong continuations from $K(\alpha)^c$ which can be constructed from w by inserting iterations of sub-LPOs into the prefix of w. For example, $I(b(2a)) = \{b^n(2a) \mid n \in \mathbb{N}\}$. The left side of Figure 18 shows example places.

A place $p_{\mathbf{r}}$ prohibits each wrong continuations in $I(w)$ (for some wrong continuation w), if and only if the following two properties are satisfied:

- $p_{\mathbf{r}}$ prohibits w.
- If the prefix of w contains the prefix of a sub-LPO which can be iterated, then the occurrence of this sub-LPO does not increase the number of tokens in $p_{\mathbf{r}}$.

The first property can be encoded as a homogenuous linear inequation as in the case of finite languages. Together with the defining property of regions of LPO-terms, the

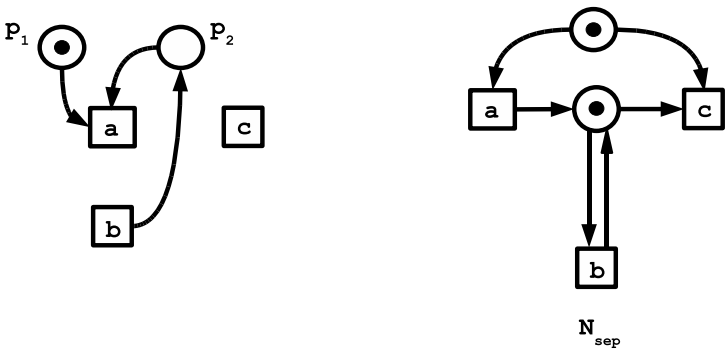


Fig. 18. Left Side: Place p_2 prohibits $b(2a)$ but not $bb(2a)$; Place p_1 prohibits each wrong continuation of the form $b^n(2a)$ for $n \in \mathbb{N}$. Right side: The separation representation of the example LPO-term.

second property implies that after the occurrence of such a sub-LPO the number of tokens in $p_{\mathbf{r}}$ is the same as before the occurrence. This can be ensured by requiring

$$Prod(lpo, p_{\mathbf{r}}) = 0$$

for each sub-LPO lpo whose prefix is prefix of the prefix of w and which can be iterated. The corresponding defining linear inequation system for transition-regions and token flow-regions can be constructed in an analoguous way as for the property $Prod(lpo, p_{\mathbf{r}}) \geq 0$ of regions of LPO-terms. For the running example LPO-term and the wrong continuation $b(2a)$ we have $I(b(2a)) = \{b\}$ and require $Prod(b, p_{\mathbf{r}}) = W(b, p_{\mathbf{r}}) - W(p_{\mathbf{r}}, b) = 0$.

It remains to determine the term of an *iterated sub-LPO*. Of course, each sub-LPO which is isomorphic to an LPO in $I(\alpha)$ is a candidat. But the follower marking after the occurrence of some sub-LPO $(V', <, l)$ only depends on the multiset $L(V')$, i.e. on the number of occurrences of each transition in the sub-LPO.

Definition 22 (Iterated sub-LPO). *We say that a sub-LPO $(V', <', l')$ of an LPO in $R(\alpha)$* can be iterated*, if $l'(V') = l(V)$ for an LPO $(V, <, l) \in I(\alpha)$.*

The right side of Figure 18 shows the separation representation for the example LPO-term. Note that the wrong continuations of the form $b^n c$ cannot be prohibited (compare also Figure 17 showing the basis representation). The reason is that b^n as well as c can occur in the initial state and b^n does not decrease the number of tokens in places. Note that it is possible to consider more general Petri net classes, as for example inhibitor nets, which allow to prohibit $b^n c$.

2.6 Speeding Up Synthesis for Finite Languages

If a finite language of LPOs contains finite iterations of sub-LPOs, then the technique from the previous subsection can be used to reduce the runtime for computing the separation representation. In the following we will illustrate this by an example.

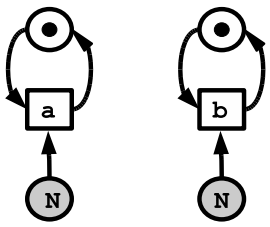


Fig. 19. PT-net synthesized from $L = \{a^N \parallel b^N\}$

Consider the familiy of LPOs $(a^n \parallel b^n)$ $(n \in \mathbb{N})$. The number of wrong continuations of the language $L = \{(a^n \parallel b^n)\}$ (for some fixed n) grows quadratically with n. For a fixed N the wrong continuations are of the form $(a^j \parallel b^i)(2a)$, $(a^j \parallel b^i)(2b)$, $(a^j \parallel b^i)(2a + b)$ and $(a^j \parallel b^i)(2b + a)$ for $0 \leq i, j \leq N - 1$, $(a^N \parallel b^i)a$, $(a^N \parallel b^i)(2b)$ and $(a^N \parallel b^i)(2b + a)$ for $0 \leq i \leq N - 1$, $(a^j \parallel b^N)b$, $(a^j \parallel b^N)(2a)$ and $(a^j \parallel b^N)(2a + b)$ for $0 \leq j \leq N - 1$, and $(a^N \parallel b^N)a$ and $(a^N \parallel b^N)b$.

Instead of considering all these wrong continuations we can equivalently

- first compute the wrong continuations of the LPO-term $(a^* \parallel b^*)$, which are only those of $a \parallel b$ (iterations only occur once in the representation set. The corresponding separation representation prohibits all wrong continuations of $L = \{(a^N \parallel b^N)\}$ except $(a^N \parallel b^i)a$ for $0 \leq i \leq N - 1$, $(a^j \parallel b^N)b$ for $0 \leq j \leq N - 1$, and $(a^N \parallel b^N)a$ and $(a^N \parallel b^N)b$.
- second add the remaining wrong continuations, which are not yet prohibited.

Figure 19 shows the PT-net synthesized with this technique. The white places allow arbitrary iterations and prohibit the first kind of wrong continuations. The grey places restrict the length of LPO-runs and prohibit the second kind of wronf continuations.

The technique significantly reduces the number of wrong continuations we need to consider.

2.7 Concluding Remarks

The presented synthesis theory for LPO-languages was developed within the last years. In earlier years, the synthesis problem already was solved for finite and regular languages over single action names. As already mentioned, such languages are a special case of LPO-languages as considered in this paper.

The technique of computing the separation representation of the set of PT-net transition regions as presented in this paper coincides with the earlier developed technique for this special case [10,2,3]. In these early publications a parametric definition of Petri nets, which can be instantiated with different concrete net classes like elementary nets or inhibitor nets, was considered. In the second part of this paper we extend the presented framework to several of these other net classes.

In [16] PT-net transition-regions for trace languages (step languages) are introduced. The authors do not consider the subclasses of finite or regular step languages and therefore only define an infinite separating representation (involving infinite many constraints)

Token flow regions and the basis representation were developed later in the context of LPO-languages.

There are many other publications considering (step) transition systems instead of languages as behavioral model (e.g., [14,15,28,29,8,30,11]). These approaches are restricted to transition-regions and separating representations.

3 Extensions and Generilizations

In this second part of the paper we extend and generalize the framework presented in the first part in the following directions:

- We consider the synthesis of inhibitor nets from finite and from simple infinite languages of labelled stratified order structures.
- We discuss synthesis from languages of non-transitive order structures.
- We examine the synthesis of nets of restricted net classes.
- We suggest several possibilities for a finite representation of more general infinite languages.

For a clear presentation we do not combine these generalizations to common definitions of regions and finite representations. Instead we consider each generilization separately and give some hints on how to combine different concepts in each subsection.

3.1 Inhibitor Nets

As examples in the first part of this paper illutrated, not always the given LPO-language can be exactly represented by a PT-net. In such cases, it is possible to consider more general Petri net classes in order find a better representation. One of the most general Petri net classes are inhibitor nets, which are as powerful as Turing machines for languages over single action names. In this subsection we extend the concept of regions to inhibitor nets.

Stratified Order Structures. Partial orders are used in the first part of this paper to represent causal dependencies between transition occurrences of PT-nets. When considering inhibitor nets, we need finer causal structures, so called *relational structures* [18]. A *relational structure* (*rel-structure*) is a triple $\mathcal{S} = (V, \prec, \sqsubset)$, where V is a finite set (of *nodes*) and $\prec \subseteq V \times V$ and $\sqsubset \subseteq V \times V$ are binary relations on V. The notions of preset and postset are only used w.r.t. the relation $\prec$. A rel-structure $\mathcal{S}$ is called *acyclic* if

$$(\prec \cup \sqsubset)^* \circ \prec \circ (\prec \cup \sqsubset)^*$$

is irreflexive. Similar to the notion of the transitive closure of a binary relation the *transitive closure* $\mathcal{S}^+$ of a rel-structure $\mathcal{S} = (V, \prec, \sqsubset)$ is defined by

$$\mathcal{S}^+ = (V, \prec_{\mathcal{S}^+}, \sqsubset_{\mathcal{S}^+}) = (V, (\prec \cup \sqsubset)^* \circ \prec \circ (\prec \cup \sqsubset)^*, (\prec \cup \sqsubset)^* \setminus id_V).$$

An *extension* of an acyclic rel-structure $(V, \prec, \sqsubset)$ is an acyclic rel-structure $(V, \prec', \sqsubset')$ satisfying $\prec \subseteq \prec' \wedge \sqsubset \subseteq \sqsubset'$. A *prefix* of an acyclic rel-structure $(V, \prec, \sqsubset)$ is an acyclic rel-structure $(V', \prec, \sqsubset)$ with $V' \subseteq V$ satisfying $(v' \in V') \wedge (v(\sqsubset \cup \prec)v') \Rightarrow (v \in V')$.

A rel-structure $\mathcal{S} = (V, \prec, \sqsubset)$ is called *stratified order structure* (*so-structure*) if the following conditions are satisfied for all $u, v, w \in V$:

- $u \not\sqsubset u$.
- $u \prec v \implies u \sqsubset v$.
- $u \sqsubset v \sqsubset w \wedge u \neq w \implies u \sqsubset w$ (weak transitivity).
- $u \sqsubset v \prec w \vee u \prec v \sqsubset w \implies u \prec w$ (strong mixed transitivity).

$(V, \prec)$ is a partial order. Thus a partial order can always be interpreted as an so-structure with $\sqsubset = \prec$. The *transitive closure* $\mathcal{S}^+$ of a rel-structure $\mathcal{S}$ is an so-structure if and only if $\mathcal{S}$ is acyclic (for this and further results see [18]). Later on, we will interpret the relation $\prec$ as an "earlier than"-relation between events and the relation $\sqsubset$ as an "not later than"-relation between events.

Two nodes $v, v' \in V$ of an so-structure $(V, \prec, \sqsubset)$ are called *independent* if $v \not\sqsubset v'$ and $v' \not\sqsubset v$. Co-sets, cuts, minimal and maximal events are defined w.r.t. the partial order $\prec$.

The *skeleton* of an so-structure $(V, \prec, \sqsubset)$ is the rel-structure $(V, \prec', \sqsubset')$ with $\prec' \subseteq \prec$ minimal, $\sqsubset' \subseteq \sqsubset$ minimal and $(V, \prec', \sqsubset')^+ = (V, \prec, \sqsubset)$.

Graphically, "earlier than"-relation is drawn by drawn-through arrows and the "not later than"-relation by dotted arrows between events. For a clear illustration, often transitiv arrows are not drawn. Figure 20 shows an example so-structure, where the nodes v_2 and v_3 are in "not later than"-relation and the nodes v_1 and v_2 as well as v_1 and v_3 are in "earlier than"-relation.

Semantics of Inhibitor Nets. Inhibitor nets are PT-nets extended by read arcs testing places for absence of tokens.

Definition 23 (Inhibitor Net). *An* inhibitor net *(*PTI-net*)* $N = (P, T, F, W, I)$ *consists of a PT-net* (P, T, F, W) *and a mapping* $I : P \times T \to \mathbb{N}_0 \cup \{\infty\}$ *called* inhibitor function.

We denote $Und(N) = (P, T, F, W)$ *the PT-net* underlying N.

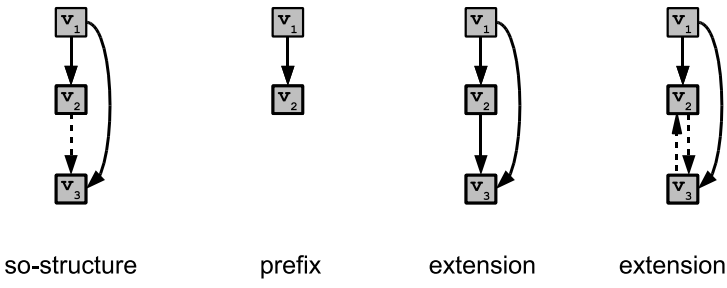


Fig. 20. An so-structure with a prefix and two extensions

The inhibitor function specifies upper bounds for the number of tokens allowed in a place for the occurrence of a transition.

Graphically, the number $I(p,t)$ is assigned to an arrow from p to t which has a circle as arrowhead, where in the case $I(p,t) = \infty$ no arrow is drawn and in the case $I(p,t) = 0$ only the arrow is drawn. Figure 22 shows a marked PTI-net with $I(p_5, b) = 0$ and $I = \infty$ in all other cases.

We introduce the following multiset of places:

- ${}^{-}t(p) = I(p,t)$ for transitions t.
- ${}^{-}\tau(p) = min(\{\, {}^{-}t(p) \mid t \in \tau\})$ for multisets of transitions τ.

There are two different semantics of inhibitor nets. We only consider the *a-priori* semantics here, because it leads to more general causal semantics. For the so called *a-posteriori* semantics of inhibitor nets, causal semantics is based on LPOs as for PT-nets.

Definition 24 (A-Priori Occurrence Rule). *A multiset of transitions* τ can occur *in* m, *if* $m \geq {}^{\bullet}\tau$ *and* $m \leq {}^{-}\tau$.

The notion of $m \xrightarrow{\tau} m'$ *is defined as for PT-nets.*

The notions of sequential executions and step executions are deduced from the occurrence rule as for PT-nets.

The PTI-net shown in Figure 22 has the sequential executions a, c, ac, ab, abc and the additional step execution $(1a, 0b, 1c), (1a, 0b, 0c)(0a, 1b, 1c)$ in the initial marking.

Finally, we recall process semantics of PTI-nets. The problem of defining processes for PTI-nets is that the absence of tokens in a place – this is tested by inhibitor arcs – cannot be directly represented in an occurrence net. This is solved by introducing local extra conditions and read arcs – also called *activator arcs* – connected to these conditions. These extra conditions are introduced "on demand" to directly represent dependencies of events caused by the presence of an inhibitor arc in the net. The conditions are artificial conditions without a reference to inhibitor weights or places of the net. They only focus on the dependencies that result from inhibitor tests. Thus, activator arcs represent local information regarding the lack of tokens in a place.

The process definition is based on the usual notion of occurrence nets extended by activator arcs. These occurrence nets are (labeled) acyclic nets with non-branching conditions whose underlying causal relationship between events is described by so-structures (similar to partial orders describing causal relationships between events of PT-net processes). In the following definition B represents the finite set of *conditions*, E the finite set of *events*, R the *flow relation* and Act the set of *activator arcs* of the occurrence net.

Definition 25 (Activator Occurrence Net [21]). *An* activator occurrence net *(ao-net) is a five-tuple* $AON = (B, E, R, Act)$ *satisfying:*

- B *and* E *are finite disjoint sets.*
- $R \subseteq (B \times E) \cup (E \times B)$ *and* $Act \subseteq B \times E$.
- *The rel-structure* $\mathcal{S}(AON) = (E, \prec_{loc}, \sqsubset_{loc}, l|_E) = (E, (R \circ R)|_{E\times E} \cup (R \circ Act), (Act^{-1} \circ R) \setminus id_E, l|_E)$ *is acyclic.*
- $\forall b \in B(|\, {}^{\bullet}b| \leq 1 \wedge |b^{\bullet}\,| \leq 1$.

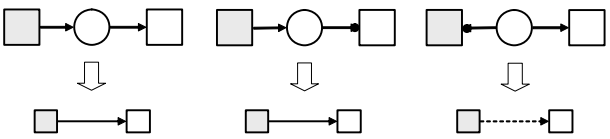


Fig. 21. Generation of the orders $\prec_{loc}$ (drawn-through arrow) and $\sqsubset_{loc}$ (dotted arrow) in ao-nets. Activator arcs have a filled cirlce as arrowhead.

For $x \in E$ and $X \subseteq E$ we denote $x^+ = \{y \mid (y,x) \in Act\}$ and $X^+ = \bigcup_{x \in X} x^+$.

Note that this definition is a conservative extension of standard occurrence nets by read arcs. The relations $\prec_{loc}$ and $\sqsubset_{loc}$ represent the local information about causal relationships between events. Figure 21 shows their construction rule.

Since an activator occurrence net can be identified with an acyclic directed graph, we can use notations introduced for acyclic directed graphs. For the definition of processes we need the notions of weak configurations and weak slices for ao-nets (there is also the notion of strong slices which we do not need in this paper). A set of events $D \subseteq E$ is called a *weak configuration* of AON, if $e \in D$ and $f(\prec_{loc} \cup \sqsubset_{loc})^+ e$ implies $f \in D$. A *weak slice* of AON is a maximal (w.r.t. set inclusion) set of conditions $S \subseteq B$ which are $R \circ (\prec_{loc} \cup \sqsubset_{loc})^* \circ R$-independent. $WSL(AON)$ denotes the set of all weak slices.

Each weak slice is of the form $S_C = (C^\bullet \cup Min(AON)) \setminus {}^\bullet C$ for a weak configuration C [21]. For a weak slice S there is always a finite sequence of steps of $\prec_{loc}^+$-independent events $\tau_1 \ldots \tau_n$ with $S \xrightarrow{\tau_1} S_1 \xrightarrow{\tau_2} \ldots \xrightarrow{\tau_n} Max(AON)$. This means weak slices represent reachable markings which allow to complete a given process.

Now we are prepared to define processes of PTI-nets as in [20]. The mentioned artificial conditions in such processes are labeled by the special symbol $\lightning$. They are introduced in two kinds of situations:

- A transition $t \in T$ tests a place in the pre- or post-multiset of another transition $w \in T$ for absence of tokens, i.e. $I(p,t) \neq \infty$ and ${}^\bullet w(p) + w^\bullet(p) \neq 0$ for some $p \in P$. Then occurrences f of w and e of t in a process must eventually be ordered via a $\lightning$-condition intended either to ensure that tokens are consumed earlier than the test occurs (${}^\bullet w(p) \neq 0$) or to ensure that tokens are produced not later than the test occurs ($w^\bullet(p) \neq 0$). Such situations are abbreviated by $w \multimap t$.
- A transiton z testing some place for absense of tokens occurs concurrently to transitions t consuming and w producing tokens in this place, i.e. $I(p,z) \neq \infty$ and $p \in {}^\bullet t \cap w^\bullet$ for some $p \in P$. Then occurrences f of w and e of t in a process must eventually be ordered via a $\lightning$-condition intended to ensure that tokens are consumed not later than produced in order to restrict the maximal number of tokens in the place according to the inhibitor weight.

In both situations the two occurrences f and e are adjacent to a common $\lightning$-condition representing the described causal dependency of f and e. This means there exists a $\lightning$-labeled condition b such that $(b,e) \in Act$ and $b \in ({}^\bullet f \cup f^\bullet)$. This is abbreviated by $f \multimap\!\bullet\, e$.

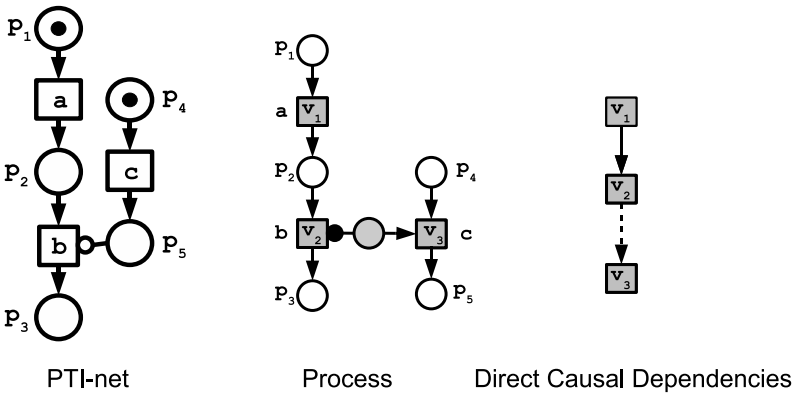


Fig. 22. A PTI-net and one of its processes

Definition 26 (Complete activator process [20]). *Let $N = (P, T, F, W, I, m_0)$ be a PTI-net. A* complete activator process *(ca-process) of N is an ao-net $AON = (B \uplus \widetilde{B}, E, R, Act)$ together with a labelling function $l : B \uplus \widetilde{B} \cup E \to P \cup T \cup \{\curlywedge\}$ satisfying:*

(Cond1) $l(B) \subseteq P$, $l(E) \subseteq T$ *and* $l(\widetilde{B}) = \{\curlywedge\}$.
(Cond2) $\widetilde{B} = \{b \mid \exists e \in E((b,e) \in Act)\}$.
(Cond3) $m_0 = l(Min(AON) \cap B)$.
(Cond4) *For all* $e \in E$: $^\bullet l(e) = l(\,^\bullet e \cap B)$ *and* $l(e)^\bullet = l(e^\bullet \cap B)$.
(Cond5) *For all* $b \in \widetilde{B}$, *there are unique* $g, h \in E$ *such that one of the following properties hold:*
- $^\bullet b \cup b^\bullet = \{g\}$, $(b,h) \in Act$ *and* $l(g) \multimap l(h)$.
- $b^\bullet = \{g\}$, $(b,h) \in Act$ *and additionally* $^\bullet l(h) \cap l(g)^\bullet \cap \,^- z \neq \emptyset$ *for some* $z \in T$.

(Cond6) *For all* $e, f \in E$: *if* $f \multimap\!\bullet\, e$ *then there is exactly one* $c \in \widetilde{B}$ *such that* $f \multimap\!\bullet\, e$ *through* c.
(Cond7) *For all* $e \in E$ *and* $S \in WSL(AON)$: *if* $^\bullet e \cup \{b \in \widetilde{B} \mid (b,e) \in Act\} \subseteq S$ *then* $l(S \cap B) \leq \,^- l(e)$.

Figure 22 shows a process of a PTI-net, where names of conditions are omitted. The names of events are shown inside, the labels of events and conditions outside of the graphical object. There is one condition from $\widetilde{B}$, which is filled by grey color. Observe that transition b tests the post-set $c^\bullet$ of c for absence of tokens, i.e. $c \multimap b$. This is reflected in the process by $v_3 \multimap\!\bullet\, v_2$. The right side of the Figure shows the acyclic rel-structure underlying the process.

The requirements (Cond1), (Cond3), (Cond4) represent common features of processes well-known from PT-nets. They ensure that ca-processes constitute a conservative extension of standard PT-net processes. This means, the set of processes of $Und(N)$ can be derived from the set of ca-processes by omitting the $\curlywedge$-labeled

conditions (omitting the λ-conditions from a ca-process AON leads to the so called *underlying process* $Und(AON)$ of AON). If N has no inhibitor arcs (thus $N = Und(N)$), ca-processes coincide with standard processes.

The properties (Cond2) and (Cond5) together with the rule (Cond6) – describing when λ-conditions have to be inserted – constitute the structure of the λ-conditions. The requirement (Cond7) expresses that in the weak slices of AON the inhibitor constraints of the PTI-net have to be properly reflected. That means, for events enabled in a certain weak slice of AON the respective transitions are also enabled in the respective marking in the PTI-net N.

If AON is a process of a PTI-net N then the underlying causal relationsships between events $\mathcal{S}(AON)^+$ form an so-structure, because $\mathcal{S}(AON)$ is an acyclic rel-structure. This means, we can represent single (non-sequential) runs of PTI-nets by so-structures labelled by transition names. Such labelled so-structure extend LPOs by adding a second causal relation between transition occurrences which is interpreted as a "not later than"-relation. This relation represents the situation that two transition occurrences can be observed simultaneously and as a sequence in one order (but not the other order). In this model of runs, we can distinguish concurrency from *synchronicity*. Synchronicity means that transiton occurrences can be observed only simultaneously, but not as a sequence in any order. If two transition occurrences are in symmetric "not later than"-relation, then they are synchronous.

Definition 27 (Labelled SO-Structure). *A* labelled so-structure *(*LSO*) over T is a 4-tuple $(V, \prec, \sqsubset, l)$, where $(V, \prec, \sqsubset)$ is an so-structure and $l : V \to T$ is a labelling function on V.*

We only consider LSOs up to isomorphism, i.e. only the labelling of events is of interest, but not the event names. Formally, two LSOs $(V, \prec, \sqsubset, l)$ and $(V', \prec', \sqsubset', l')$ are *isomorphic*, if there is a renaming function $I : V \to V'$ satisfying $l(v) = l'(I(v))$, $v \prec w \Leftrightarrow I(v) \prec' I(w)$ and $v \sqsubset w \Leftrightarrow I(v) \sqsubset' I(w)$.

As a special kind of LSOs we consider *linear LSOs*. A linear LSO is an LSO satisfying $co_\prec = \sqsubset \setminus \prec$, This means, the relation $\sqsubset \setminus \prec$ is symmetric and defines synchronous transition occurrences. The maximal sets of synchronous transition occurrences are called *synchronous steps*. The synchronous steps of a linear LSO are linearly ordered w.r.t. the "earlier than"-relation. Linear LSOs cannot be extended and represent (synchronous) step executions of PTI-nets.

The *set of step linearizations* of an LSO is the set of linear LSOs which are extensions of this LSO. For example, lso_1 shown in Figure 23 is not linear, since two events are in asymetric "not later than"-relation. The LSOs lso_2 and lso_3 are step-linearizations of lso_1, where

- lso_2 represents the step execution $(1a, 0b, 0c)(0a, 1b, 0c)(0a, 0b, 1c)$,
- lso_3 represents the step execution $(1a, 0b, 0c)(0a, 1b, 1c)$.

Definition 28 (LSO-run). *An LSO $(V, \prec, \sqsubset, l)$ is an* LSO-run *of a PTI-net N if there is a ca-process AON of N such that $(V, \prec, \sqsubset, l)$ is an extension of $\mathcal{S}(AON)$.*

An LSO-run lso is said to be minimal*, if there exists no other LSO-run lso' of N such that lso is an extension of lso'.*

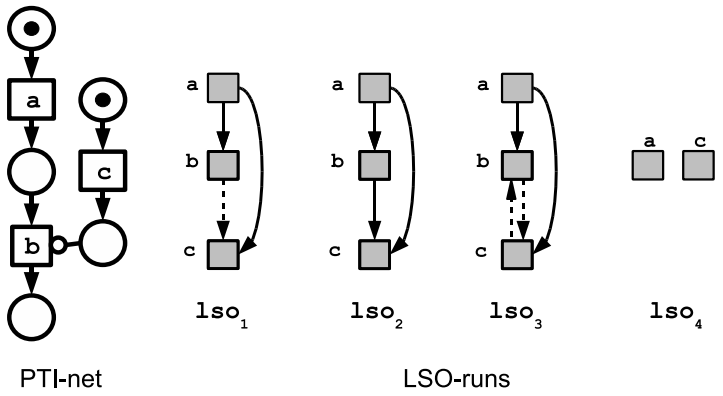


Fig. 23. A PTI-net with four of its LSO-runs. The LSOs lso_2 and lso_3 are step-linearizations of lso_1. The LSO-runs lso_1 and lso_4 are minimal.

Figure 23 shows a PTI-net together with some of its LSO-runs. Note that the LSO-run lso_1 exactly represents all causal dependencies between transition occurrences of a process of the net (which is shown in Figure 22). Moreover, lso_1 is minimal, since b may not occur simultaneously with a and may not occur after c.

From the definition follows that extensions of LSO-runs also are LSO-runs. This means, the set of all LSO-runs can be deduced from the set of minimal LSO-runs.

We show in [20] that an LSO $lso = (V, \prec, \sqsubset, l)$ is an LSO-run of a PTI-net N if and only if each step-linearization of lso is a step execution of N. Equivalently, lso is an LSO-run if and only if for each cut C of lso and each place p of $N = (P, T, F, W, I, m_0)$ there holds:

$$m_0(p) + \sum_{v<C} (W(l(v), p) - W(p, l(v))) \geq \sum_{v \in C} W(p, l(v))$$

and

$$m_0(p) + \sum_{v<C} (W(l(v), p) - W(p, l(v))) \leq I(p, t)$$

for each $t \in l(C)$.

We often omit transitive arrows of LSOs for a clearer presentation.

Regions of PTI-Nets. The formal problem statement, which we consider from now, is:

Given: A prefix-closed and extension-closed finite language L of LSOs over a finite alphabet of transition names T.

Searched: A PTI-net N with set of transitions T such that all LSOs in L are LSO-runs of N and N has a minimal number of additional LPO-runs.

As for PT-nets and LPOs, we define transition regions and token flow regions of PTI-nets as as non-negative integral solutions of appropriate linear systems of the form

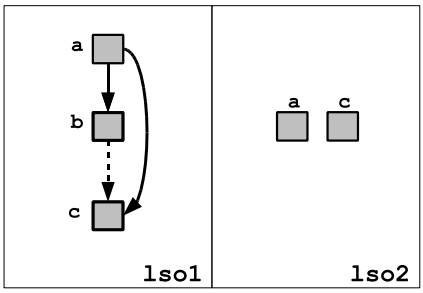


Fig. 24. Running example language (prefixes and extensions are not shown)

$\mathbf{A}_L \cdot \mathbf{x} \leq \mathbf{b}_L$. As in the case of LPOs, it is enough to consider only those LSOs from L, which are not extensions or prefixes of other LSOs from L. Throughout the rest of this subsection we use the language represented by the LSOs shown in Figure 24 as a running example.

A *(PTI-net) transition-region* $\mathbf{r}$ directly defines the parameters of a place $p_{\mathbf{r}}$ of PTI-nets, i.e. it determines the numbers $m_0(p_{\mathbf{r}})$, $W(p_{\mathbf{r}}, t)$ and $W(t, p_{\mathbf{r}})$ for each $t \in T$ (PT-net part), and $I(p_{\mathbf{r}}, t)$ for each $t \in T$. If $T = \{t_1, \ldots, t_n\}$, then $\mathbf{r}$ is given as a $(3n+1)$-tuple $\mathbf{r} = (r_0, \ldots, r_{3n})$ of non-negative integers. Its components define these numbers via $m_0(p_{\mathbf{r}}) = r_0$, $W(p_{\mathbf{r}}, t_i) = r_i$, $W(t_i, p_{\mathbf{r}}) = r_{n+i}$ and $I(p_{\mathbf{r}}, t_i) = r_{n+2i}$ for $i \in \{1, \ldots, n\}$. In the running example, denote $t_1 = a$, $t_2 = b$ and $t_3 = c$.

Since a region $\mathbf{r}$ is intended to define a *feasible* place $p_{\mathbf{r}}$, it is required to satisfy a property $(f)_L$ ensuring that $p_{\mathbf{r}}$ is feasible w.r.t. L. Remember that $p_{\mathbf{r}}$ is feasible w.r.t. L if the net resulting from adding $p_{\mathbf{r}}$ still generates at least L. For this, the property $(f)_L$ formalizes that

- (as in the PT-net case) for each cut of events there are at least as much tokens in $p_{\mathbf{r}}$ as consumed by the occurrence of the corresponding step of transitions after the occurrence of the prefix preceeding the cut (PT-net constraint).
- additionally for each cut of events there are at most as much tokens in $p_{\mathbf{r}}$ as required by inhibitor tests of transitions in the corresponding step of transitions after the occurrence of the prefix preceeding the cut (inhibitor constraint).

In the running example transition step $(1b, 1c)$ of $lso1$ must be able to occur after one occurrence of a. This means, $p_{\mathbf{r}}$ has to satisfy

- $m_0(p_{\mathbf{r}}) - W(p_{\mathbf{r}}, a) + W(a, p_{\mathbf{r}}) \geq W(p_{\mathbf{r}}, b) + W(p_{\mathbf{r}}, c)$,
- $m_0(p_{\mathbf{r}}) - W(p_{\mathbf{r}}, a) + W(a, p_{\mathbf{r}}) \leq I(p_{\mathbf{r}}, b)$,
- $m_0(p_{\mathbf{r}}) - W(p_{\mathbf{r}}, a) + W(a, p_{\mathbf{r}}) \leq I(p_{\mathbf{r}}, c)$,

i.e. $r_0 - r_1 + r_4 \geq r_2 + r_3$, $r_0 - r_1 + r_4 \leq r_8$ and $r_0 - r_1 + r_4 \leq r_9$.

The property $(f)_L$ for a finite language L of LSOs and PTI-nets contains all PT-net constraints and additionally the following *inhibitor constraint*: For each $lso = (V, <, \sqsubset, l) \in L$, for each cut C of lso and for each $t = t_k \in l(C)$:

$$r_0 + \sum_{i=1}^{n} l(V')(t_i)(r_{n+i} - r_i) - r_{2n+k} \leq 0,$$

where $V' = \{v \in V \mid v \prec C\}$. This is the case if and only if $\mathbf{b}_{lso,C,t} \cdot \mathbf{r} \leq 0$ for the vector $\mathbf{b}_{lso,C,t} = (b_{C,t,0}, \ldots, b_{C,t,3n})$ defined by

$$\mathbf{b}_{C,t,j} = \begin{cases} 1 & \text{if } j = 0, \\ -l(V')(t_j) & \text{if } j \in \{1, \ldots, n\}, \\ l(V')(t_{j-n}) & \text{if } n+1 \leq j \leq 2n, \\ -1 & \text{if } j = 2n+k, \\ 0 & \text{else} \end{cases}$$

For the cut C corrsponding to the transition step $(1b, 1c)$ of $lso1$ in the running example we require $r_0 - r_1 + r_4 \leq r_8$ and $r_0 - r_1 + r_4 \leq r_9$. This is the case if and only if $\mathbf{b}_{lso1,C,b} \cdot \mathbf{r} \leq 0$ and $\mathbf{b}_{lso1,C,c} \cdot \mathbf{r} \leq 0$ for

$$\mathbf{b}_{lso1,C,b} = (1, -1, 0, 0, 1, 0, 0, 0, -1, 0),$$

$$\mathbf{b}_{lso1,C,c} = (1, -1, 0, 0, 1, 0, 0, 0, 0, -1).$$

Definition 29 (Transition-Region). *A tuple* $\mathbf{r}$ *as above is called a* transition-region *if it satisfies* $(f)_L$.

For the defined property $(f)_L$ the following theorem holds [27]:

Theorem 9. *A tuple* $\mathbf{r}$ *satisfies* $(f)_L$ *if and only if* $p_{\mathbf{r}}$ *is feasible w.r.t.* L.

Let $\mathbf{A}_L$ be the matrix consisting of all rows $\mathbf{a}_{lso,C}$ (PT-net constraints) and $\mathbf{b}_{lso,C,t}$ for LSOs $lso = (V, <, \sqsubset, l) \in L$, cuts C of lso and transitions $t \in l(C)$. Since L is assumed to be finite, $\mathbf{A}_L$ is finite. Thus, the set of all regions can be computed as the set of all integral solutions of the homogenous linear inequation system $\mathbf{A}_L \cdot \mathbf{x} \leq \mathbf{0}$.

Note that we never compute the inhibitor weight ∞ representing the case that there is no inhibitor restriction. This is not necessary in the case of a finite LSO-language, since each feasible place is bounded (for each feasible place p there is an upper $b \in \mathbb{N}$ such that $m(p) \leq b$ for all reachable markings m). In this case, an inhibitor weight exceeding the bound for the number of tokens in a place is equivalent to ∞, i.e. does not restrict the behavior. After the computation of a feasible place, it can be simplified by replacing such useless inhibitor weights by the value ∞.

All places of the PTI-net from Figure 22 are solutions for the running example language. For example, the region $r_2 = (0, 0, 1, 0, 1, 0, 0, 1, 1, 1)$ defines p_2 and the region $r_5 = (0, 0, 0, 1, 0, 0, 0, 1, 0, 1)$ defines p_5 (these regions still contain the useless inhibitor value 1 which can be replaced by ∞, since the corresponding places are 1-bounded).

Theorem 10 ([27]). *If* L *is finite then there is a finite matrix* $\mathbf{A}_L$ *such that the set of transition-regions is the set of solutions of the linear inequation system* $\mathbf{A}_L \cdot \mathbf{x} \leq \mathbf{0}$.

A *token flow-region* $\mathbf{r}$ defines a place $p_{\mathbf{r}}$ indirectly by determining the *token flow* w.r.t. this place between transition occurrences in LSOs from L, i.e. by directly determining the number of tokens produced by a transition occurrence which are consumed by a subsequent transition occurrence in an LSO specified in L. Such numbers are assigned to the "earlier than"-arrows between transition occurrences of LSOs. As for LPOs, for each transition occurrence the number of tokens consumed from the initial marking, the number of tokens which are produced but not further consumed by other transition occurrences and the number of tokens in the initial marking which are not consumed by any transition occurrence of an LSO are considered.

Since inhibitor values cannot be represented by token flows, we define them directly in the same way as for transition regions. If $W = \bigcup_{(V,\prec,\sqsubset,l)\in L} V$ is the set of nodes of LSOs in L and $E = \bigcup_{(V,\prec,\sqsubset,l)\in L} \prec$ is the set of "earlier than"-arrows of LSOs in L, then a *(PTI-net) token flow-region* $\mathbf{r}$ is given as a tuple $\mathbf{r} = (r_i)_{i\in W\times\{in,out\}\cup E\cup L\cup T}$ of non-negative integers. The components r_i with $i \in W \times \{in, out\} \cup E \cup L$ are interpreted as in the PT-net case and $I(p_{\mathbf{r}}, t) = r_t$ for $t \in T$. Initial marking and the weight function w.r.t. $p_{\mathbf{r}}$ are defined as in the PT-net case.

Since a region $\mathbf{r}$ is intended to define a *feasible* place $p_{\mathbf{r}}$, it is required to satisfy a property $(wd)_L$ ensuring that $p_{\mathbf{r}}$ is feasible w.r.t. L. The property $(wd)_L$ for PTI-nets and LSOs requires addionally to the PT-net constraints that the marking reached after the occurrence of some prefix of an LSO in L does not exceed the inhibitor constraints of transition occurrences subsequent to this prefix, i.e. for each $lso = (V, \prec, \sqsubset, l) \in L$, for each cut C of lso and for each $t \in l(C)$:

$$r_{lso} + \sum_{v\in V\setminus V'} r_{v,in} + \sum_{v'\in V',\, v\in V\setminus V'} r_{(v',v)} + \sum_{v'\in V'} r_{v',out} - r_t \leq 0,$$

where $V' = \{v \in V \mid v \prec C\}$. This is the case it and only if $\mathbf{A}_{L,d} \cdot \mathbf{r} \leq \mathbf{0}$, where for each t and each cut C of an LSO $lso \in L$ with $t \in l(C)$ and $V' = \{v \in V \mid v \prec C\}$ the matrix $\mathbf{A}_{L,d}$ has a row $\mathbf{d}_{C,t} = (d_{C,t,i})_{i\in W\times\{in,out\}\cup E\cup L\cup T}$ defined by

$$d_{n,t,i} = \begin{cases} 1 & \text{if } i = lso, \\ 1 & \text{if } i = (v, in) \wedge v \in V \setminus V', \\ 1 & \text{if } i = (v', v) \wedge v' \in V' \wedge v \in V \setminus V', \\ 1 & \text{if } i = (v', out) \wedge v' \in V', \\ -1 & \text{if } i = t \\ 0 & \text{else.} \end{cases}$$

Definition 30 (Token Flow-Region). *A tuple* $\mathbf{r}$ *as above is called a* token flow-region *if it satisfies* $(wd)_L$.

For the property $(wd)_L$ the following theorem holds for PTI-net places:

Theorem 11 ([5]). *A tuple* $\mathbf{r}$ *satisfies* $(wd)_L$ *if and only if* $p_{\mathbf{r}}$ *is feasible w.r.t.* L.

Let $\mathbf{A}_L$ be the matrix consisting of all rows from the matrices $\mathbf{A}_{L,a}$, $\mathbf{A}_{L,b}$, $\mathbf{A}_{L,c}$ (PT-net constraints) and $\mathbf{A}_{L,d}$ (inhibitor constraint). Since L is assumed to be finite, $\mathbf{A}_L$ is finite. Thus, the set of all token flow-regions can be computed as the set of all integral solutions of the homogenous linear equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$.

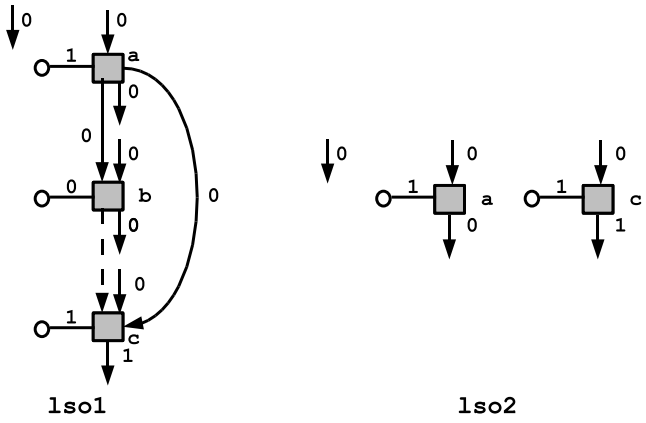


Fig. 25. Illustration of a PTI-net token flow region

All places of the PTI-net from Figure 22 are solutions for the running example language. Figure 10 shows a token flow region representing the place p_5 from Figure 22. Inhibitor weights are annotated to additional arcs having a circle as arrowhead. Again, useless inhibitor weights exceeding a place bound can be replaced by the value ∞. Observe that the token flow leaving the prefix consisting of the occurrence of a of $lso1$ equals 0 such that the inhibitor constraint for the subsequent occurrence of b is fulfilled.

Theorem 12 ([5,27]). *If L is finite then there is a finite matrix $\mathbf{A}_L$ such that the set of token flow-regions is the set of solutions of the linear equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$.*

Finite Representations of PTI-Net Regions. As in the case of PT-nets and LPO-languages, the set of PTI-net regions of an LSO-language is defined as the set of positive integral solutions of a homogenuous linear inequation system. Thus, it has a finite basis representation. Many places of this basis represenation are relatively complex, since there is an inhibitor arc connection to every transition. As argued in the last paragraph, these inhibitor weight may be useless. In this case a place can be simplified by replacing the useless inhibitor weight by the value ∞. As in the case of PT-nets, many basis places are implicit and can be omitted and in some cases there are easy strategies to compute them. Figure 27 shows some implicit places for the running example in grey color. A detailed examination is a topic of future research.

In order to compute a finite separation representation we need to compute the set L^c of wrong continuations of an LSO-language L. As for LPO-languages we denote by L^{step} the set of step linearizations of LSOs in L. We need to consider two kinds of wrong continuations: Wrong continuations of an analoguous form as in the LPO-case and wrong continuations representing situations where steps of transitions cannot be sequentialized in any order.

Definition 31 (Wrong Continuation of an LSO-language). *A* wrong flow continuation *of an LSO-language L is a sequence of transition steps of the form $w_{\sigma,t}$*

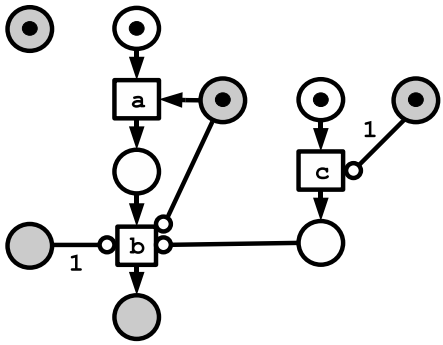


Fig. 26. Some basis solution places for the running example. Iimplicit places are filled with grey color.

$= \alpha_1 \dots \alpha_{n-1}(\alpha_n + t) \notin L^{step}$ *for* $\sigma = \alpha_1 \dots \alpha_{n-1}\alpha_n \in L^{step}$ *and* $t \in T$*, where we call* $\alpha_1 \dots \alpha_{n-1}$ *the* prefix *and* $\alpha_n + t$ *the* follower step of the wrong continuation.

A wrong inhibitor continuation *of an LSO-language* L *is a sequence of transition steps of the form* $w_{\sigma,\beta_1,\beta_2} = \alpha_1 \dots \alpha_{n-1}\beta_1\beta_2 \notin L^{step}$ *for* $\beta_1 + \beta_2 \leq \alpha_n$ *and* $\sigma = \alpha_1 \dots \alpha_{n-1}\alpha_n \in L^{step}$*, where we call* $\alpha_1 \dots \alpha_{n-1}\beta_1$ *the* prefix *and* β_2 *the* follower step of the wrong continuation.

Some of the wrong flow continuations of the running example are $(2a), b, 2c$ (all having an empty prefix) and one wrong inhibitor continuation of the running example is acb (since $a(b + c) \in L^{step}$), where multisets are denoted as sums of singleton multisets.

As in the case of LPO-languages, we represent a wrong flow continuation by a pair of multisets $(l(V'), l(S \cup \{z\}))$, where $(V', \prec, \sqsubset, l)$ is a prefix of an LSO in L representing the prefix of the wrong flow continuation, S is a subset of direct successors of $(V', \prec, \sqsubset, l)$ and z is an additional labelled event, where S and z together represent the follower step of the wrong flow continuation.

A wrong inhibitor continuation $\alpha_1 \dots \alpha_{n-1}\beta_1\beta_2$ we represent by a pair of multisets $(l(V' \cup B_1), l(B_2))$, where $(V', \prec, \sqsubset, l)$ is a prefix of an LSO in L representing $\alpha_1 \dots \alpha_{n-1}$, B_1 is a subset of direct successors of this prefix representing β_1 and B_2 is a subset of direct successors of this prefix representing β_2.

To prohibit a wrong flow continuation, one needs to find a feasible place p such that one of the following constraints is fulfilled:

- *PT-net constraint*: After occurrence of its prefix there are not as much tokens in p as its follower step consumes.
- *Inhibitor constraint*: After occurrence of its prefix there are more tokens in p as allowed by the inhibitor constraint of the additional event in the follower step. Note that this constraint only can be fulfilled, if the label of the additional event does not occur twice in the follower step.

To prohibit a wrong inhibitor continuation $\alpha_1 \dots \alpha_{n-1}\beta_1\beta_2$, one needs to find a feasible place p such that the following inhibitor constraint is fulfilled: After occurrence of its prefix there are more tokens in p as allowed by the inhibitor constraints of the events

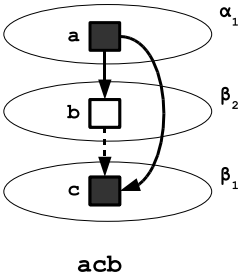


Fig. 27. Illustration of the wrong inhibitor continuation acb

in the follower step. Note that, since $\alpha_1 \dots \alpha_{n-1}\alpha_n \in L^{step}$ and $\beta_1 + \beta_2 \leq \alpha_n$, after occurrence of $\alpha_1 \dots \alpha_{n-1}\beta_1$ there are always at least as many tokens in a place p as β_2 consumes.

The PT-net constraint can be expressed as a linear inequality as in the case of LPO-languages and PT-nets. In the following we show, that also the inhibitor constraints of a wrong continuation can represented by a linear inequality.

If $\mathbf{r}$ is a transition region, $T = \{t_1, \dots, t_n\}$ and $l(z) = t_k$ then $p_{\mathbf{r}}$ satisfies the inhibitor constraint for a wrong flow continuation if and only if

$$r_0 + \sum_{i=1}^{n} l(V')(t_i)(r_{n+i} - r_i) - r_{2n+k} > 0.$$

This is the case if and only if $\mathbf{d}(w_{\sigma,t_k}) \cdot \mathbf{r} < 0$ for $\mathbf{d}(w_{\sigma,t_k}) = (d_0, \dots, d_{3n})$ defined by:

$$d_j = \begin{cases} -1 & \text{if } j = 0, \\ l(V')(t_j) & \text{if } j \in \{1, \dots, n\}, \\ -l(V')(t_{j-n}) & \text{if } n+1 \leq j \leq 2n, \\ 1 & \text{if } j = 2n+k, \\ 0 & \text{else} \end{cases}$$

Similarly, if $\mathbf{r}$ is a transition region and $T = \{t_1, \dots, t_n\}$ then $p_{\mathbf{r}}$ satisfies the inhibitor constraint for a wrong inhibitor continuation if and only if, for each k with $t_k \in \beta_2$, $\mathbf{d}(w_{\sigma,t_k}) \cdot \mathbf{r} < 0$ for $\mathbf{d}(w_{\sigma,t_k}) = (d_0, \dots, d_{3n})$ defined as for wrong flow continuations. For example, for the wrong inhibitor continuation acb we require $-r_0 + ((r_1 - r_4) + 2(r_3 - r_6)) + (r_8) < 0$ (remember $t_1 = a$, $t_2 = b$ and $t_3 = c$). This is the case if and only if $\mathbf{d}(w_{\sigma,t}) \cdot \mathbf{r} < 0$ for

$$\mathbf{d}(w_{\sigma,t}) = (-1, 1, 0, 1, -1, 0, -1, 0, 1, 0).$$

The region

$$\mathbf{r} = (0, 0, 0, 0, 0, 0, 1, 1, 0, 1)$$

(corresponding to place p_5 in Figure 22 after replacing useless inhibitor weights by ∞) is a solution which prohibits this wrong continuation.

If $\mathbf{r}$ is a token flow region and $l(z) = t$ then $p_{\mathbf{r}}$ satisfies the inhibitor constraint if and only if

$$r_{lso} + \sum_{u \notin V'} r_{u,in} + \sum_{v \in V'} r_{v,out} + \sum_{v \in V',\, u \notin V'} r_{v,u} - r_t > 0.$$

This is the case if and only if $\mathbf{d}(w_{\sigma,t}) \cdot \mathbf{r} < 0$ for $\mathbf{d}(w_{\sigma,t}) = (d_i)_{i \in W \times \{in,out\} \cup E \cup L \cup T}$ defined by:

$$d_i = \begin{cases} -1 & \text{if } i = lso, \\ -1 & \text{if } i = (u, in) \wedge u \notin V', \\ -1 & \text{if } i = (v, out) \wedge v \in V', \\ -1 & \text{if } i = (v, u) \wedge v \in V' \wedge u \notin V', \\ 1 & \text{if } i = t, \\ 0 & \text{else.} \end{cases}$$

Similarly, if $\mathbf{r}$ is a token flow region then $p_{\mathbf{r}}$ satisfies the inhibitor constraint if and only if, for $t \in \beta_2$, $\mathbf{d}(w_{\sigma,t}) \cdot \mathbf{r} < 0$ for $\mathbf{d}(w_{\sigma,t}) = (d_i)_{i \in W \times \{in,out\} \cup E \cup L \cup T}$ defined as for wrong flow continuations. The token flow region shown in Figure 25 prohibits the wrong continuation acb.

One possible strategy for computing a token flow regions prohibiting a wrong flow continuation is:

- First try to find a solution using the PT-net constraint.
- If there is no solution using the PT-net constraint, then try to find a solution using the inhibitor constraint.

In order to compute a token flow region prohibiting a wrong inhibitor continuation, there is only to possibility to use the inhibitor constraint. For example, the wrong continuation acb cannot be prohibited by a place satisfying the PT-net constraint, since $a(b + c) \in L^{step}$.

There are techniques for computing simple places, as for example:

- If the PT-net constraint is used, all inhibitor weights can be chosen as the value ∞. A target function can be used to minimize initial marking and weight on flow arrows as in the case of LPO-languages.
- If the inhibitor constraint is used, through an appropriate target function the inhibitor weight associated to forbidden transitions can be chosen minimal (all other inhibitor weights can be chosen as the value ∞).
- The number of wrong inhibitor continuations $\alpha_1 \ldots \alpha_{n-1}\beta_1\beta_2$ can be reduced by considering only singleton multisets β_1 and β_2 (since bigger multisets always contain such a singleton).

Infinite LSO-Languages. It is possible to extend the presented framework to infinite LSO-languages along the same lines as in the case of LPO-languages. The idea is to use LPO-terms extended by a term-based representation of "not later than"-relations.

To represent "not later than"-relations between events we introduce a new composition operator $<$ for *weak sequential composition*. *Synchronous composition* between events will be expressed by a new operator $<>$.

Since a term like $(a; b) <> (c; d)$ cannot be interpreted in a meaningful way, we do not apply $<>$ to arbitrary terms, but only to single action names. On the other hand, $<$ can be applied to arbitrary terms: Writing $\alpha < \beta$ for terms α, β means that all events in α are in "not later than"-relation to all events in β.

We do not introduce an operator for the iteration of the operator $<$ for the same reason we do not consider iteration of the operator $\|$: Such iteration operators allow to specify runs with arbitrary large multisets of synchronous resp. independent transition occurrences in the same state of the system (a state of the system can be identified with each prefix of a specified run). Such a behavior cannot be produced through Petri net models.

Let A be a finite set of LSOs. For $A \in \mathcal{A}$ we write $A = (V_A, \prec_A, \sqsubset_A, l_A)$. We denote by $\lambda = (\emptyset, \emptyset, \emptyset, \emptyset)$ the empty LSO. LSOs consisting only of one single event we denote by the label of this event.

Definition 32 (LSO-term). *The set of* LSO-terms *over a finite set of LSOs $\mathcal{A}$ is inductively defined as follows:*

- *Each singleton LSO from $\mathcal{A}$ is a* synchronous step.
- *If s_1 and s_2 are synchronous steps, then $s_1 <> s_2$ is a* synchronous step.
- *The elements $A \in \mathcal{A}$, all synchronous steps and λ are* LSO-terms.
- *Let α_1 and α_2 be* LSO-terms. *Then*
  - $\alpha_1; \alpha_2$ *(sequential composition),*
  - $\alpha_1 < \alpha_2$ *(weak sequential composition),*
  - $\alpha_1 + \alpha_2$ *(union),*
  - $(\alpha_1)^*$ *(iteration),*
  - $\alpha_1 \parallel \alpha_2$ *(parallel composition)*

  are LSO-terms.

In the following we consider the running example LSO-term $c^*; (a < b)$. Figure 28 illustrates the LSO-language generated by this term.

We assign to an arbitrary LSO-term α a possibly infinite prefix and extension closed LSO-language $L(\alpha)$. The language $L(\alpha)$ is defined as the prefix and extension closure of an appropriate LSO-language $K(\alpha)$. In order to construct $K(\alpha)$, we define the weak sequential composition of two LSOs A and B by

$$A < B = (V_A \cup V_B, <_A \cup <_B \cup (V_A \setminus Max(A) \times V_B) \cup (V_A \times V_B \setminus Min(B)),$$
$$\sqsubset_A \cup \sqsubset_B \cup (V_A \times V_B), l_A \cup l_B).$$

Sequential composition, parallel composition and iteration of LSOs is defined w.r.t. $\prec$ as for LPOs. We identify synchronous steps with LSOs as follows: If $s_A = (V_A, \prec_A, \sqsubset_A, l_A)$ and $s_B = (V_B, \prec_B, \sqsubset_B, l_B)$ are synchronous steps then

$$s_A <> s_B = (V_A \cup V_B, \emptyset, \sqsubset_A \cup \sqsubset_B \cup (V_A \times V_B) \cup (V_B \times V_A), l_A \cup l_B).$$

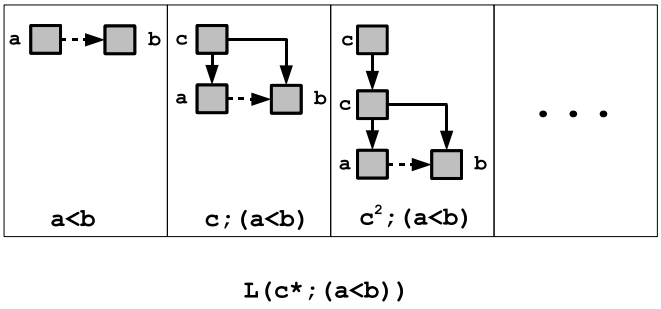


Fig. 28. Infinite LSO-language, represented by an LSO-term

Definition 33 (LSO-language of an LSO-term). *We define inductively:*

- $K(\lambda) = \{\lambda\}$, $K(A) = \{A\}$ *for* $A \in \mathcal{A}$ *and* $K(s) = s$ *for synchronous steps* s.
- *Let* α_1 *and* α_2 *be LSO-terms. Then:*
  - $K(\alpha_1 + \alpha_2) = K(\alpha_1) \cup K(\alpha_2)$,
  - $K(\alpha_1; \alpha_2) = \{A_1 A_2 \mid A_1 \in K(\alpha_1),\ A_2 \in K(\alpha_2)\}$,
  - $K(\alpha_1 < \alpha_2) = \{A_1 < A_2 \mid A_1 \in K(\alpha_1),\ A_2 \in K(\alpha_2)\}$,
  - $K((\alpha_1)^*) = \{A_1 \dots A_n \mid A_1, \dots, A_n \in K(\alpha_1)\} \cup \{\lambda\}$,
  - $K(\alpha_1 \parallel \alpha_2) = \{A_1 \parallel A_2 \mid A_1 \in K(\alpha_1),\ A_2 \in K(\alpha_2)\}$.

Some of the LSOs in $K(c^*; (a < b))$ are $a < b$, $c; (a < b)$ and $c^2; (a < b)$.

Since the additional operations do not introduce side effects to iterations, $K(\alpha)$ can be finitely represented by a representation set $R(\alpha)$ and an iteration set $I(\alpha)$ in an analoguous way as in the case of LPO-terms:

Definition 34 (Representation/Iteration Set of LSO-terms). *The* representation set $R(\alpha)$ *and the* iteration set $I(\alpha)$ *of an LSO-term* α *are defined inductively as follows* (α_1 *and* α_2 *are LSO-terms,* $\mathcal{S}$ *denotes the set of synchronous steps):*

$R(\lambda) = \{\lambda\}$	$I(\lambda) = \emptyset$
$R(A) = \{A\}$ *for* $A \in \mathcal{A}$	$I(A) = \emptyset$ *for* $A \in \mathcal{A}$
$R(s) = \{s\}$ *for* $s \in \mathcal{S}$	$I(s) = \emptyset$ *for* $s \in \mathcal{S}$
$R(\alpha_1 + \alpha_2) = R(\alpha_1) \cup R(\alpha_2)$	$I(\alpha_1 + \alpha_2) = I(\alpha_1) \cup I(\alpha_2)$
$R(\alpha_1; \alpha_2) = \{AB \mid A \in R(\alpha_1), B \in R(\alpha_2)\}$	$I(\alpha_1; \alpha_2) = I(\alpha_1) \cup I(\alpha_2)$
$R(\alpha_1 < \alpha_2) = \{A < B \mid A \in R(\alpha_1), B \in R(\alpha_2)\}$	$I(\alpha_1 < \alpha_2) = I(\alpha_1) \cup I(\alpha_2)$
$R((\alpha_1)^*) = R(\alpha_1) \cup \{\lambda\}$	$I((\alpha_1)^*) = I(\alpha_1) \cup R(\alpha_1)$
$R(\alpha_1 \parallel \alpha_2) = \{A \parallel B \mid A \in R(\alpha_1), B \in R(\alpha_2)\}$	$I(\alpha_1 \parallel \alpha_2) = I(\alpha_1) \cup I(\alpha_2)$

In the running example, $R(c^*; (a < b)) = \{a < b, c; (a < b)\})$ and $I(c^*; (a < b)) = \{c\}$.

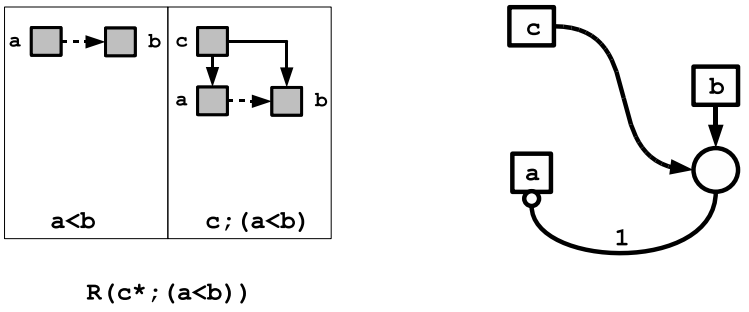


Fig. 29. A non-feasible place

We start to characterize the set of feasible places of $L(\alpha)$ analoguously as for LPO-terms by defining regions $\mathbf{r}$ of LSO-terms as regions w.r.t. the finite set $R(\alpha)$ satisfying the additional property

$$Prod(lso, p_{\mathbf{r}}) := \sum_{t \in l(V)} l(V)(t)(W(t, p_{\mathbf{r}}) - W(p_{\mathbf{r}}, t)) \geq 0.$$

for all LSOs $lso = (V, \prec, \sqsubset, l) \in I(\alpha)$. In the running example this means $(W(c, p_{\mathbf{r}}) - W(p_{\mathbf{r}}, c)) \geq 0$.

But the set of places corresponding to such regions still contains places which are not feasible. For example, the place shown in Figure 29 is feasible w.r.t. $R(c^*; (a < b))$, but prohibits $c^2; (a < b)$, because iterations of c add tokens to a place with an inhibitor restriction w.r.t. a. The general situation is a place p with the following properties:

- There is an LSO $lso \in I(\alpha)$ with $Prod(lso, p) > 0$ (in the example $lso = c$).
- There is an event v of an LSO in $(V, \prec, \sqsubset, l) \in R(\alpha)$ such that:
  - $I(p, l(v)) < \infty$ (in the example $I(p, a) < \infty$).
  - The prefix of a sub-LSO isomorphic to lso is prefix of the prefix of a cut containing v (the empty prefix in the example).

If these properties are satisfied, p is not feasible because the iteration of lso arbitrarily increases the number of tokens in p such that in the end the occurrence of $l(v)$ is prohibited by $I(p, l(v)) < \infty$. Obviously, we can construct a feasible place from p by changing $I(p, l(v))$ into the value ∞ in each such situation. If we do this for all such places p, we get the set of all places which are feasible w.r.t. $L(\alpha)$. Some of these places can be simplyfied by replacing useless inhibitor weights by the value ∞ as in the finite case, if the place is bounded and the inhibitor weight exceeds the place bound. Bounded places p can be easily found, since they are characterized by $Prod(lso, p) = 0$ for all LSOs $lso \in I(\alpha)$.

Since the set of regions again can be defined as the set of solutions of a linear homogenuous inequation system, it is possible to generate a basis representation. Basis places which are not feasible can be turned into feasible places and simplified as described above.

It is also possible to compute a finite separation representation using the following definition of wrong continuations which combines ideas from wrong continuations of LPO-terms and finite LSO-languages.

Definition 35 (Wrong Continuation of LSO-term). *Let α be an LPO-term and let $\sigma = \beta_1 \dots \beta_{n-1}\beta_n \in R(\alpha)^{step}$ and $t \in T$ such that $w_{\sigma,t} = \beta_1 \dots \beta_{n-1}(\beta_n + t) \notin K(\alpha)^{step}$, where β_n is allowed to be the empty step. Then $w_{\sigma,t}$ is called* wrong flow continuation of α. *We call $\beta_1 \dots \beta_{n-1}$ the* prefix *and $\beta_n + t$ the* follower step of the wrong flow continuation.

A wrong inhibitor continuation *of an LSO-term α is a sequence of transition steps of the form $w_{\sigma,\beta_1,\beta_2} = \alpha_1 \dots \alpha_{n-1}\beta_1\beta_2 \notin R(\alpha)^{step}$ for $\beta_1 + \beta_2 \leq \alpha_n$ and $\sigma = \alpha_1 \dots \alpha_{n-1}\alpha_n \in R(\alpha)^{step}$, where we call $\alpha_1 \dots \alpha_{n-1}\beta_1$ the* prefix *and β_2 the* follower step of the wrong inhibitor continuation.

In the example, some wrong flow continuations are $(a + b)a, c(a + b)b, 2c$ and some wrong inhibitor continuations are ba, cba. As for finite LSO-languages, wrong flow continuations can be forbidden using a PT-net constraint or an inhibitor constraint, wrong inhibitor continuations can be only forbidden using an inhibitor constraint. If the PT-net constraint is used, all inhibitor weights can be chosen to be ∞ and for some iterated LSOs lso we require $Prod(lso, p_{\mathbf{r}}) = 0$ as in the case of LPO-terms. If the inhibitor constraint w.r.t. an event v is used, we also require $Prod(lso, p_{\mathbf{r}}) = 0$ for some iterated LSOs lso, namely if the prefix of a sub-LSO isomorphic to lso is prefix of the prefix of a cut containing an event with label $l(v)$ (as argued above).

Each LPO-term is a special case of an LSO-term. This means, if the synthesis problem of LPO-term has no exact PT-net solution, then can try find an exact PTI-net solution using the extended techniques from this paragraph. For example, consider the LPO-term $((a; b^*) \parallel b^*) + c$ from the subsection on LPO-terms. As discribed, it is not possible to prohibit wrong continuations of the form $b^n c$ by a PT-net place. As illustrated in Figure 30 it is possible to prohibit these wrong continuations by a PTI-net place.

3.2 Non-transitive Causal Structures

It is possible to specify the set of runs of a Petri net by non-transitive causal structures like labelled acyclic directed graphs (for PT-nets) or labelled acyclic rel-structures (for PTI-nets). In the following we only consider labelled acyclic directed graphs in more detail. All definitions and results can be extended and generalized to labelled acyclic rel-structures along the same lines as before for LPOs and LSOs.

Acyclic directed graphs labelled by transition names can be used to represent the direct causal dependencies between transition occurrences underlying processes. This means, we require a token flow between transition occurrences along each specified arrow.

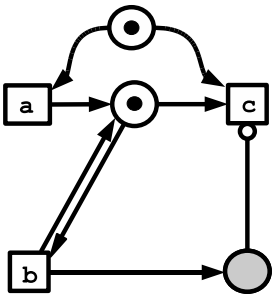


Fig. 30. The grey PTI-net place prohibits executions of the form $b^n c$

Definition 36 (Labelled Acyclic Directed Graph). *A* labelled acyclic directed graph *(*LDAG*)* over T *is a 3-tuple* $(V, \rightarrow, l)$*, where* $(V, \rightarrow)$ *is an acyclic directed graph and* $l : V \rightarrow T$ *is a labelling function on* V*.*

As LPOs, we only consider LDAGs up to isomorphism.

Definition 37 (LDAG-run). *An LDAG* $(V, \rightarrow, l)$ *is a* LDAG-run *of a PT-net* $N = (P, T, F, W, m_0)$ *if there is a process* $K = (O, \rho)$*,* $O = (B, E, G)$*, of* N *such that* $(V, \rightarrow) = (E, \{(e, f) \mid e^\bullet \cap {}^\bullet f \neq \emptyset\})$ *and* $l = \rho|_E$*.*

An LDAG-run $ldag$ *of* N *is said to be* minimal*, if there exists no other LDAG-run* $ldag'$ *of* N *such that* $ldag$ *is an extension of* $ldag'$*.*

From the definition follows that extensions of LDAG-runs in general are no LDAG-runs (see Figure 31).

The formal synthesis problem statement, which we consider here, is:

Given: A language L of LDAGs over a finite alphabet of transition names T.

Searched: A PT-net N with set of transitions T such that all LDAGs in L are LDAG-runs of N and N has a minimal number of additional minimal LDAG-runs.

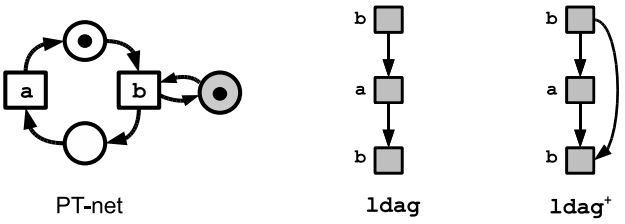


Fig. 31. For the PT-net without the grey place, $ldag$ is an LDAG-run, but $ldag^+$ is not an LDAG-run. For the PT-net with the grey place, $ldag^+$ is an LDAG-run, but $ldag$ is not.

On the one side, it is not clear how to define transition regions for LDAG-languages, since direct causal dependencies can be expressed by the components of transition regions.

On the other side, a token flow region contains components directly representing all direct causal dependencies within a specified run. This means, we can define token flow regions of LDAG-languages in an analoguous way as for LPO-languages. If between two transition occurrences there is no arrow specified, then no token flow is possible between these transition occurrences (compare $ldag$ and the PT-net without the grey place in Figure 31). If between two transition occurrences there is an arrow specified, then always a token flow region with positive token flow along this edge can be found (compare $ldag^+$ and the PT-net with the grey place in Figure 31).

The basis representation of the set of token flow regions of an LDAG-language is defined as for LPO-languages. By construction, the basis representation generates all specified LDAG-runs.

In order to define a separating representation of an LDAG-language L, observe that if an LDAG $ldag$ is an LDAG-run of a PT-net, then its transitive closure $ldag^+$ is an LPO-run of the PT-net. This means conversely, if $ldag^+$ is not an LPO-run, then $ldag$ is not an LDAG-run. Thus, each wrong continuation of $L^+ = \{ldag^+ \mid ldag \in L\}$ is also a wrong continuation of L. The computation of the separating representation is analoguous as in the case of LPOs by representing wrong continuations on the level of LDAGs. It remains to ensure that for each specified arrow there is a place such that there is token flow along this arrow w.r.t. this place, if this is not yet the case. Such places can be computed by token flow regions through requiring a positive token flow on such an arrow edge through an appropriate homogenuous linear inequation. For example, consider $ldag^+$ from Figure 31: The white places of the shown PT-net separate all wrong continuations $(2b), a, bb, b(2a), ba(2b), baa, babb, baba$, while the grey place ensures the direct causal dependency between the two occurrences of transition b specified by the transitive edge in $ldag^+$.

In order to define inifinite LDAG-languages, LDAG-terms can be defined analoguosly as LPO-terms. The only difference concerns the definition of sequential composition AB of LDAGs $A = (V_A, \rightarrow_A, l_A)$ and $B = (V_B, \rightarrow_B, l_B)$ used for the generation of the language of an LDAG-term. Since LDAGs represent only direct causal dependencies between transition occurrences, the sequential composition only introduces such direct causal dependencies between maximal events of the first and minimal events of the second LDAG:

$$AB = (V_A \cup V_B, <_A \cup <_B \cup (Max(A) \times Min(B), l_A \cup l_B).$$

3.3 Restricted Net Classes

In this subsection we discuss the synthesis of nets from several restricted net classes. In principle, each restriction which can be encoded as a finite set of linear inequations over the components of regions can be integrated into the defintion of regions. This way, regions can represent places of restricted net classes. In the following paragraphs we briefly suggest several such restrictions.

Some of the restrictions leed to non-homogenuous linear inequations of the form $\mathbf{B} \cdot \mathbf{x} \leq \mathbf{b}$ with $\mathbf{b} \neq \mathbf{0}$. In these cases, the set of solutions has no basis representation, i.e. the separation representation must be computed.

Bounds for Place Markings. For finite languages, each prefix of a specified causal structure represents a reachable marking. The number of tokens in a place reached after occurrence of such a prefix can be expressed as a linear sum of components of transition regions and token flow regions. Thus a bound restricting the number of tokens in all places can be introduced by a set of non-homogenuous linear inequations (for each prefix a linear inequation must be added).

For infinite languages, all prefixes of the finite representation set need to be considered. Additionally, iterated parts may not increase the number of tokens in a place.

Using these ideas, places of bounded PT-nets and bounded PTI-nets can be computed.

Bounds for Flow Weights. Also flow weigths can be can be expressed as a linear sum of components of transition regions and token flow regions:

- Transition regions have components directly representing flow weights.
- For token flow regions, the intoken flows and outtoken flows represent flow weights.

This means, a bound restricting all flow weights can be introduced by a set of non-homogenuous linear inequations.

Using the bound 1 for markings and flow weights, it is possible to compute places of elementary nets.

Bounds for Inhibitor Weights. Analoguously to flow weights, inhibitor weights can be bounded, because they are directly represented by components of transition regions and token flow regions.

Using the bound 0 for inhibitor weights, it is possible to compute places of simple inhibitor nets.

Final Markings. A final marking is a marking reached after occurrence of a complete specified causal structure. It is possible to introduce combinations of the following useful restrictions for final markings by sets of linear inequations:

- All final markings of some subset of specified causal structures are equal (homogenuous linear inequations).
- The final marking of some causal structure is bounded (non-homogenuous linear inequations).
- The final marking of some causal structure equals a fixed number.

For infinite languages, in the second and third case, all final markigs of the finite representation set need to be considered. Additionally, iterated parts may not increase the number of tokens in a place.

Initial Markings. It is possible to introduce combinations of the following useful restrictions for initial markings by sets of linear inequations:

- The initial marking of all places is bounded (non-homogenuous linear inequations).
- The initial marking of all places is a fixed number.

It is possible to compute intermediate places of sound workflow nets by requiring that all initial and final markings equal the number 0.

3.4 More on Infinite Languages

LPO-terms (and LSO-terms) only represent a small class of LPO-languages. In the following we show several examples of languages which cannot be represented by LPO-terms and discuss possibilities to generalize LPO-terms in order to cover some of these examples.

Figure 33 shows a simple example of a PT-net-language which cannot be represented by an LPO-term. The reason for this is that by the iteration operator it is not possible to append LPOs to a part of a previous LPO, but only to the whole LPO.

Therefore *generalized LPO-terms*, allowing to iteratively append LPOs only to parts of previous LPOs, are introduced in [7]. These parts, which we call *interfaces*, are given by prefixes containing maximal events of the LPO. In the example, the LPO $a;(b \parallel c)$ is iterated only w.r.t. its prefix $a;b$. This can be expressed by operators $;_X$ and ^{*X} for sequential composition and iteration w.r.t. an interface X.

Figure 33 shows a simple safe Petri net, whose LPO-language even cannot be represented by such generalized LPO-terms. This is because for no choice of an iterated part there is a prefix of the iterated part, such that all subsequent events causally depend on all events from this prefix. Instead, different subsequent events depend on different prefixes.

One possibility to further generalize interfaces is to specify direct causal dependencies between events of the previous LPO and events of the subsequent LPO by pairs of multisets of action names.

The PT-net-language in the last example then can be represented by the term $((a \parallel c);_X (d \parallel b))^{*Y}$ with

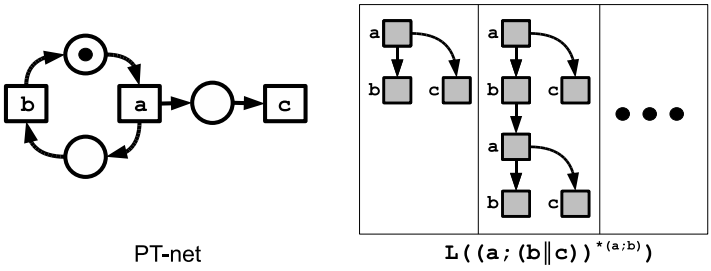


Fig. 32. A PT-net whose set of LPO-runs cannot be represented by an LPO-term

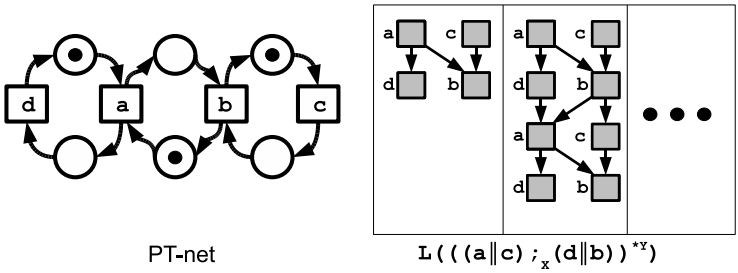


Fig. 33. A PT-net whose set of runs cannot be represented by a generalized LPO-term.

- an interface set $X = \{(a, b + d), (a + c, b)\}$ defining which events of the second LPO are directly causally dependent on which events of the previous LPO.
- and an interface set $Y = \{(d + b, a), (b, a + c)\}$ defining which events of the subsequent occurrence of the iterated LPO are directly causally dependent on which events of the previous occurrence of the iterated LPO.

In general an interface $X = \{(A_1, B_1), \ldots, (A_k, B_k)\}$ is interpreted as follows: If an LPO $(V, <, l) = lpo_1;_X lpo_2$ is constructed from a sequential composition of two LPOs lpo_1 and lpo_2 w.r.t. this interface, then it satisfies the following properties:

- If v is a maximal event of lpo_1 with $l(v) \in \bigcup_i A_i$ and W' is the set of its direct successor events in lpo_2, then $l(W') \leq B_i$ for some B_i with $l(v) \in A_i$.
- If w is a minimal event of lpo_2 with $l(w) \in \bigcup_i B_i$ and V' is the set of its direct predecessor events in lpo_1, then $l(V') = A_i$ for some maximal multiset A_i among $A_1, \ldots, A_n$ with $l(w) \in B_i$.

Consider the interface X in the previous example:

- An occurrence of a can be directly followed by at most one occurrence of b and one occurrence of d, since $(a, b + d) \in X$ and $b < b + d$.
- An occurrence of b exactly directly causally depends on one occurrence of a and one occurrence of c, since $(a + c, b) \in X$ and $a < a + c$.

Analoguous properties must hold for iterations w.r.t. interfaces.

Figure 34 shows that in presence of several equally labelled events there are several possibilities to sequentially compose LPOs w.r.t. to a given interface.

Note that the interfaces $X = \{(a, b+d), (a+c, b)\}$ and $X' = \{(a, b), (a, d), (c, b)\}$ have a different interpretation. According to X', an occurrence of a is not allowed to have two direct successors. Instead, alternatively action b or action d can occur directly after a. Moreover, an occurrence of b directly causally depends on an occurrence of a or an occurrence of c, but not on both occurrences.

This approach is very flexible and intuitive, since only direct causal dependencies between actions need to be specified and also multiple direct causal dependencies can be considered (for example, the interface $\{(2b, a)\}$ specufies that a directly causally

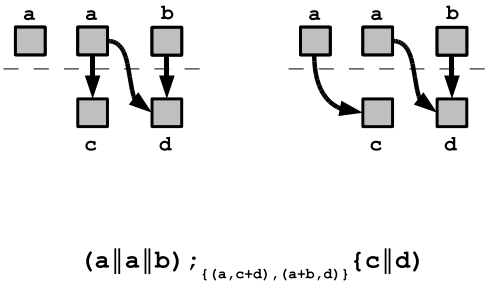


Fig. 34. Two instantiations of an interface

depends on two occurrences of b). It allows to construct arbitrary LPOs from single action names.

Moreover, it can also be applied to LSOs, as illustrated by Figure 35.

On the other side, as Figure 34 illustrates, the set of possible instantiations of an interface may be complex and difficult to predict from the syntax. Another formulation, which is more clear and restrictive, is to specify interfaces directly through LPOs.

For all mentioned generilizations through interfaces, the notions of transition region and token flow region can be adapted in such a way that they are characterized as solutions of an appropriate homogenuous linear inequations system and such that it is possible to compute a basis representation and a separation representation.

As for LPO-terms and LSO-terms, a finite representation set and a finite iteration set represent the language generated by term with interfaces.

The representation set is defined as for LPO-terms and LSO-terms, where sequential composition is defined w.r.t. a given interface. We define

$$R(\alpha_1;_X \alpha_2) = \{A;_x B \mid A \in R(\alpha_1), B \in R(\alpha_2),\ x \in R(A,B,X)\},$$

where $R(A,B,X)$ is the set of instantiations of X w.r.t. A and B (as described above and illustrated in Figure 34 - we omit a formal definition here) and

$$A;_x B = (V_A \cup V_B, <_A \cup <_B \cup x, l_A \cup l_B)$$

in the case LPO-terms with interfaces (this definition can be extended to LSO-terms with interfaces in a straightforward way).

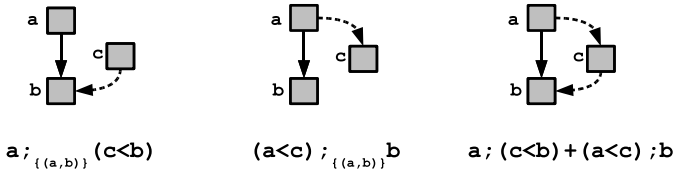


Fig. 35. Some LSOs which cannot be constructed from sinlge action names by LSO-terms

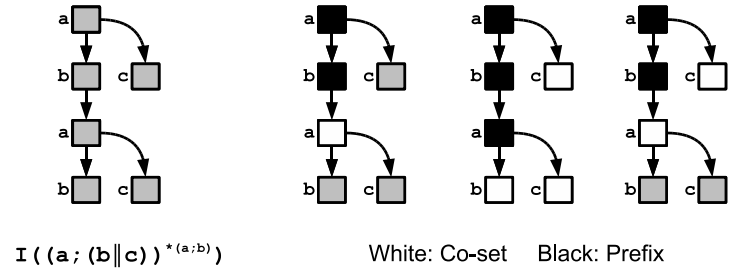


Fig. 36. A double iteration with some co-sets of events, which are considered for the definition of regions

In order to reflect interface connections, the iteration set is not defined from single iterations, but double iterations:

$$I(\alpha^{*X}) = \{A;_x B \mid A, B \in R(\alpha),\, x \in R(A, B, X)\} \cup I(\alpha).$$

For each double iteration $A;_x B$ we require that B can occur solely consuming tokens produced by A. This means that each co-set of $A;_x B$ containing events from B can occur after the occurrence of its prefix in $A;_x B$. This can be encoded by linear inequations in the usual way, where "initial markings" of double iterations are chosen consistent with markings reached after prefixes of corresponding iterated parts in $R(\alpha)$. Figure 36 illustrates some of the considered co-sets for the example shown in Figure 32.

4 Conclusion

In this paper we gave a survey on region based synthesis of Petri nets from languages. We considered place/transition nets (PT-nets), inhibitor nets (PTI-nets) and several restrictions of these net classes on the one side, and languages of labelled partial orders, labelled stratified orders, labelled acyclic graphs and labelled relational structures on the other side. The presented framework includes synthesis from finite languages and several classes of infinite languages finitely represented in term based notations and integrates all classical results on sequential languages.

Most of the results are combinations and reformulations of results from [27], [26] and [7]. There are some easy new adaptions of techniques for token flow regions to transition regions as the synthesis of PT-nets from transition regions of LPO-terms. New developments, which are not yet published, are:

- Computation of the separation representation of regions of LPO-terms.
- Computation of the separation representation of regions of finite LSO-languages.
- Definition of LSO-terms and synthesis from LSO-terms.
- Synthesis from non-transitive causal structures.
- Synthesis of nets from terms with general interfaces.

There is tool support for several of the presented techniques:

- The graphical Petri net editor VIPtool [13] supports business process modelling and synthesis and has also verification and simulation capabilities.
- The command line tool Synops [24] supports the term-based construction of partial languages and the synthesis of Petri nets from partial languages.

References

1. Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), July 10-13, Bratislava, Slovak Republic. IEEE Computer Society (2007)
2. Badouel, E., Darondeau, P.: On the Synthesis of General Petri Nets. Technical Report 3025, Inria (1996)
3. Badouel, E., Darondeau, P.: Theory of Regions. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
4. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007)
5. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of Petri Nets from Finite Partial Languages. Fundam. Inform. 88(4), 437–468 (2008)
6. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Construction of process models from example runs. In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 243–259. Springer, Heidelberg (2009)
7. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. Fundam. Inform. 95(1), 187–217 (2009)
8. Busi, N., Pinna, G.M.: Synthesis of Nets with Inhibitor Arcs. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 151–165. Springer, Heidelberg (1997)
9. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Hardware and Petri Nets: Application to Asynchronous Circuit Design. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 1–15. Springer, Heidelberg (2000)
10. Darondeau, P.: Deriving Unbounded Petri Nets from Formal Languages. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 533–548. Springer, Heidelberg (1998)
11. Darondeau, P.: Unbounded Petri Net Synthesis. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 413–438. Springer, Heidelberg (2004)
12. Desel, J.: From Human Knowledge to Process Models. In: Kaschek, R., Kop, C., Steinberger, C., Fliedl, G. (eds.) UNISCON. LNBIP, vol. 5, pp. 84–95. Springer, Heidelberg (2008)
13. Desel, J.: VipTool-Homepage (2010), `http://www.fernuni-hagen.de/se/viptool.html`
14. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-Structures. Part I: Basic Notions and the Representation Problem / Part II: State Spaces of Concurrent Systems. Acta Inf. 27(4), 315–368 (1989)
15. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-Structures. Part II: State Spaces of Concurrent Systems. Acta Inf. 27(4), 343–368 (1989)
16. Hoogers, P., Kleijn, H., Thiagarajan, P.: A Trace Semantics for Petri Nets. Information and Computation 117(1), 98–114 (1995)
17. van der Werf, C.H.J., van Dongen, B., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundamenta Informaticae 94(3), 387–412 (2009)
18. Janicki, R., Koutny, M.: Semantics of Inhibitor Nets. Inf. Comput. 123(1), 1–16 (1995)

19. Josephs, M.B., Furey, D.P.: A Programming Approach to the Design of Asynchronous Logic Blocks. In: Cortadella, J., Yakovlev, A., Rozenberg, G. (eds.) Concurrency and Hardware Design. LNCS, vol. 2549, pp. 34–60. Springer, Heidelberg (2002)
20. Juhas, G., Lorenz, R., Mauser, S.: Complete Process Semantics of Petri Nets. Fundamenta Informaticae 87(3-4), 331–365 (2008)
21. Kleijn, H.C.M., Koutny, M.: Process Semantics of General Inhibitor Nets. Inf. Comput. 190(1), 18–69 (2004)
22. Lodaya, K., Weil, P.: Series-Parallel Posets: Algebra, Automata and Languages. In: Meinel, C., Morvan, M. (eds.) STACS 1998. LNCS, vol. 1373, pp. 555–565. Springer, Heidelberg (1998)
23. Lodaya, K., Weil, P.: Series-Parallel Languages and the Bounded-Width Property. Theor. Comput. Sci. 237(1-2), 347–380 (2000)
24. Lorenz, R.: Synops-Homepage (2010), `http://www.informatik.uni-augsburg.de/lehrstuehle/inf/projekte/synops/`
25. Lorenz, R., Juhás, G., Bergenthum, R., Desel, J., Mauser, S.: Executability of Scenarios in Petri Nets. Theor. Comput. Sci. 410(12-13), 1190–1216 (2009)
26. Lorenz, R., Mauser, S., Bergenthum, R.: Theory of Regions for the Synthesis of Inhibitor Nets from Scenarios. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 342–361. Springer, Heidelberg (2007)
27. Lorenz, R., Mauser, S., Juhás, G.: How to Synthesize Nets from Languages: A Survey. In: Henderson, S.G., Biller, B., Hsieh, M.-H., Shortle, J., Tew, J.D., Barton, R.R. (eds.) Winter Simulation Conference, pp. 637–647. WSC (2007)
28. Mukund, M.: Petri Nets and Step Transition Systems. Int. J. Found. Comput. Sci. 3(4), 443–478 (1992)
29. Pietkiewicz-Koutny, M.: The Synthesis Problem for Elementary Net Systems with Inhibitor Arcs. Fundam. Inform. 40(2-3), 251–283 (1999)
30. Pietkiewicz-Koutny, M.: Synthesising Elementary Net Systems with Inhibitor Arcs from Step Transition Systems. Fundam. Inform. 50(2), 175–203 (2002)
31. van der Aalst, W.M.P., Günther, C.W.: Finding Structure in Unstructured Processes: The Case for Process Mining. In: ACSD [1], pp. 3–12
32. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow Mining: A Survey of Issues and Approaches. Data Knowl. Eng. 47(2), 237–267 (2003)
33. Zhou, M., Cesare, F.D.: Petri Net Synthesis for Discrete Event Control of Manufacturing Systems. Kluwer (1993)

Discovering Petri Nets from Event Logs

Wil M.P. van der Aalst and Boudewijn F. van Dongen

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands
{W.M.P.v.d.Aalst,B.F.v.Dongen}@tue.nl

Abstract. As information systems are becoming more and more intertwined with the operational processes they support, multitudes of events are recorded by todays information systems. The goal of *process mining* is to use such event data to extract process related information, e.g., to automatically discover a process model by observing events recorded by some system or to check the conformance of a given model by comparing it with reality. In this article, we focus on *process discovery*, i.e., extracting a process model from an event log. We focus on Petri nets as a representation language, because of the concurrent and unstructured nature of real-life processes. The goal is to introduce several approaches to discover Petri nets from event data (notably the α-algorithm, state-based regions, and language-based regions). Moreover, important requirements for process discovery are discussed. For example, process mining is only meaningful if one can deal with *incompleteness* (only a fraction of all possible behavior is observed) and *noise* (one would like to abstract from infrequent random behavior). These requirements reveal significant challenges for future research in this domain.

Keywords: Process mining, Process discovery, Petri nets, Theory of regions.

1 Introduction

Process mining provides a new means to improve processes in a variety of application domains [2, 41]. There are two main drivers for this new technology. On the one hand, more and more events are being recorded thus providing detailed information about the history of processes. Despite the omnipresence of event data, most organizations diagnose problems based on fiction rather than facts. On the other hand, vendors of Business Process Management (BPM) and Business Intelligence (BI) software have been promising miracles. Although BPM and BI technologies received lots of attention, they did not live up to the expectations raised by academics, consultants, and software vendors.

Process mining is an emerging discipline providing comprehensive sets of tools to provide fact-based insights and to support process improvements [2, 7]. This new discipline builds on process model-driven approaches and data mining. However, process mining is much more than an amalgamation of existing approaches. For example, existing data mining techniques are too data-centric to provide a

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 372–422, 2013.

comprehensive understanding of the end-to-end processes in an organization. BI tools focus on simple dashboards and reporting rather than clear-cut business process insights. BPM suites heavily rely on experts modeling idealized to-be processes and do not help the stakeholders to understand the as-is processes.

Over the last decade *event data* has become readily available and process mining techniques have matured. Moreover, process mining algorithms have been implemented in various academic and commercial systems. Examples of commercial systems that support process mining are: *ARIS Process Performance Manager* by Software AG, *Disco* by Fluxicon, *Enterprise Visualization Suite* by Businesscape, *Interstage BPME* by Fujitsu, *Process Discovery Focus* by Iontas, *Reflect|one* by Pallas Athena, and *Reflect* by Futura Process Intelligence. Today, there is an active group of researchers working on process mining and it has become one of the "hot topics" in BPM research. Moreover, there is a huge interest from industry in process mining. This is illustrated by the recently released *Process Mining Manifesto* [41]. The manifesto is supported by 53 organizations and 77 process mining experts contributed to it. The manifesto has been translated into a dozen languages (http://www.win.tue.nl/ieeetfpm/). The active contributions from end-users, tool vendors, consultants, analysts, and researchers illustrate the growing relevance of process mining as a bridge between data mining and business process modeling. Moreover, more and more software vendors started adding process mining functionality to their tools. The authors have been involved in the development of the open-source process mining tool *ProM* right from the start [11, 56, 57]. ProM is widely used all over the globe and provides an easy starting point for practitioners, students, and academics.

Whereas it is easy to discover sequential processes, it is very challenging to discover concurrent processes, especially in the context of noisy and incomplete event logs. Given the concurrent nature of most real-life processes, Petri nets are an obvious candidate to represent discovered processes. Moreover, most real-life processes are not nicely block-structured, therefore, the graph based nature of Petri nets is more suitable than notations that enforce more structure.

The article is based on a lecture given at the *Advanced Course on Petri nets* in Rostock, Germany (September 2010). The practical relevance of process discovery and the suitability of Petri net as a basic representation for concurrent processes motivated us to write this tutorial.

Figure 1 illustrates the concept of process discovery using a small example. The figure shows an abstraction of an event log. There are 1391 cases, i.e., process instances. Each case is described as a sequence of activities, i.e., a trace. In this particular log there are 21 different traces. For example, trace $\langle a, c, d, e, h\rangle$ occurs 455 times, i.e., there are 455 cases for which this sequence of activities was executed. The challenge is to discover a Petri net given such an event log. A discovery algorithm such as the α-algorithm [9] is able to discover the Petri net shown in Figure 1.

Process discovery is a challenging problem because one cannot assume that all possible sequences are indeed present. Consider for example the event log shown in Figure 1. If we randomly take 500 cases from the set of 1391 cases,

#	trace
455	acdeh
191	abdeg
177	adceh
144	abdeh
111	acdeg
82	adceg
56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg

#	trace
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg

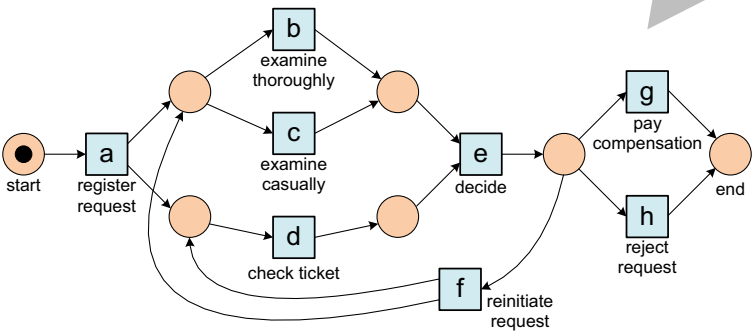


Fig. 1. A Petri net discovered from an event log containing 1391 cases

we would like to discover "more or less" the same model. Note that there are several traces that appear only once in the log. Many of these will disappear when considering a log with only 500 cases. Also note that the process model discovered by the α-algorithm allows for more traces than the ones depicted in Figure 1, e.g., $\langle a, d, c, e, f, d, b, e, f, c, d, e, h \rangle$ is possible according to the process model but does not occur in the event log. This illustrates that event logs tend to be *far from complete*, i.e., only a small subset of all possible behavior can be observed because the number of variations is larger than the number of instances observed.

The process model in Figure 1 is rather simple. Real-life processes will consist of dozens or even hundreds of different activities. Moreover, some behaviors will be very infrequent compared to others. Such rare behaviors can be seen as *noise* (e.g., exceptions). Typically, it is undesirable and also unfeasible to capture frequent and infrequent behavior in a single diagram.

Process discovery techniques need to be able to deal with noise and incompleteness. This makes process mining *very different from synthesis*. Classical synthesis techniques aim at creating a model that captures the given behavior *precisely*. For example, classical language-based region techniques [14, 17, 19, 28, 42, 43, 45] distill a Petri net from a (possibly infinite) language, such that the behavior of the Petri net is only minimally more than the given language. In classical state-based region theory [13, 15, 23, 24, 26, 27, 35] on the other hand, a transition system is used to synthesize a Petri net of which the behavior is *bisimilar* with the given transition system. Intuitively two models are bisimilar if they can match each other's moves, i.e., they cannot be distinguished from one another by an observer [36]. In terms of mining this implies that the naïvely synthesized Petri net cannot generalize beyond the example traces seen.

Process discovery techniques need to balance four criteria: *fitness* (the discovered model should allow for the behavior seen in the event log), *precision* (the discovered model should not allow for behavior completely unrelated to what was seen in the event log), *generalization* (the discovered model should generalize the example behavior seen in the event log), and *simplicity* (the discovered model should be as simple as possible). This makes process discovery a challenging and highly relevant topic.

The remainder of this article is organized as follows. Section 2 introduces the process mining spectrum showing that process discovery is an essential ingredient for process analysis based on facts rather than fiction. Section 3 presents preliminaries and formalizes the process discovery task. The α-algorithm is presented in Section 4. Section 5 discusses the main challenges related to process mining. In Section 6, we compare process discovery with region theory in more detail. This section shows that classical approaches cannot deal with particular requirements essential for process mining. Then, in sections 7 and 8, we show how region theory can be adapted to deal with these requirements. Both state-based regions and language-based regions are considered. All approaches described in this article are supported by *ProM*, the leading open-source process mining framework. ProM is described in Section 9. Section 10 ends this article with some conclusions and challenges that remain.

2 Process Mining

Process mining is an important tool for modern organizations that need to manage non-trivial operational processes. On the one hand, there is an incredible growth of event data [44]. On the other hand, processes and information need to be aligned perfectly in order to meet requirements related to compliance, efficiency, and customer service. *Process mining is much broader than just control-flow discovery*, i.e., discovering a Petri net from a multi-set of traces. Therefore, we start by providing an overview of the process mining spectrum.

Event logs can be used to conduct three types of process mining as shown in Figure 2 [2, 7].

The first type of process mining is *discovery*. A discovery technique takes an event log and produces a model without using any a-priori information.

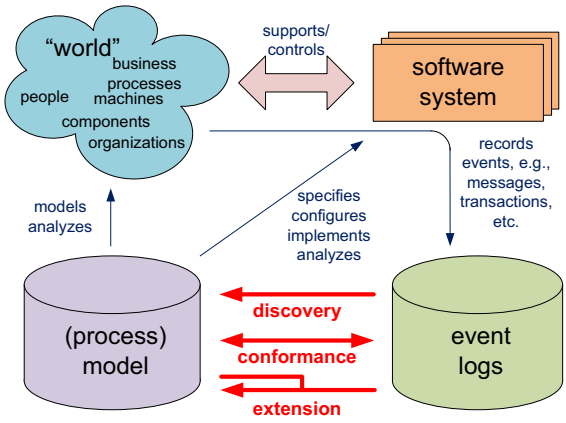


Fig. 2. Positioning of the three main types of process mining: *discovery*, *conformance*, and *enhancement*

An example is the α-algorithm [9] that will be described in Section 4. This algorithm takes an event log and produces a Petri net explaining the behavior recorded in the log. For example, given sufficient example executions of the process shown in Figure 1, the α-algorithm is able to automatically construct the Petri net without using any additional knowledge. If the event log contains information about resources, one can also discover resource-related models, e.g., a social network showing how people work together in an organization.

The second type of process mining is *conformance*. Here, an existing process model is compared with an event log of the same process. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa. For instance, there may be a process model indicating that purchase orders of more than one million Euro require two checks. Analysis of the event log will show whether this rule is followed or not. Another example is the checking of the so-called "four-eyes" principle stating that particular activities should not be executed by one and the same person. By scanning the event log using a model specifying these requirements, one can discover potential cases of fraud. Hence, conformance checking may be used to detect, locate and explain deviations, and to measure the severity of these deviations. An example is the conformance checking algorithm described in [51]. Given the model shown in Figure 1 and a corresponding event log, this algorithm can quantify and diagnose deviations. In [4] another approach based on creating *alignments* is presented. An alignment is *optimal* if it relates the trace in the log to a most similar path in the model. After creating optimal alignments, all behavior in the log can be related to the model.

The third type of process mining is *enhancement*. Here, the idea is to extend or improve an existing process model using information about the actual process recorded in some event log. Whereas conformance checking measures the alignment between model and reality, this third type of process mining aims at

changing or extending the a-priori model. One type of enhancement is *repair*, i.e., modifying the model to better reflect reality. For example, if two activities are modeled sequentially but in reality can happen in any order, then the model may be corrected to reflect this. Another type of enhancement is *extension*, i.e., adding a new perspective to the process model by cross-correlating it with the log. An example is the extension of a process model with performance data. For instance, Figure 1 can be extended with information about resources, decision rules, quality metrics, etc.

The Petri net in Figure 1 only shows the control-flow. However, when extending process models, additional perspectives need to be added. Moreover, discovery and conformance techniques are not limited to control-flow. For example, one can discover a social network and check the validity of some organizational model using an event log. Hence, orthogonal to the three types of mining (discovery, conformance, and enhancement), different perspectives can be identified. The *organizational perspective* focuses on information about resources hidden in the log, i.e., which actors (e.g., people, systems, roles, and departments) are involved and how are they related. The goal is to either structure the organization by classifying people in terms of roles and organizational units or to show the social network. The *time perspective* is concerned with the timing and frequency of events. When events bear timestamps it is possible to discover bottlenecks, measure service levels, monitor the utilization of resources, and predict the remaining processing time of running cases.

3 Process Discovery: Preliminaries and Purpose

In this section, we describe the goal of process discovery. In order to do this, we present a particular format for logging events and a particular process modeling language (i.e., Petri nets). Based on this we sketch various process discovery approaches.

3.1 Event Logs

The goal of process mining is to extract knowledge about a particular (operational) process from event logs, i.e., process mining describes a family of *a-posteriori* analysis techniques exploiting the information recorded in audit trails, transaction logs, databases, etc. Typically, these approaches assume that it is possible to *sequentially record events* such that each event refers to an *activity* (i.e., a well-defined step in the process) and is related to a particular *case* (i.e., a process instance). Furthermore, some mining techniques use additional information such as the performer or *originator* of the event (i.e., the person / resource executing or initiating the activity), the *timestamp* of the event, or *data elements* recorded with the event (e.g., the size of an order).

To clarify the notion of an event log consider Table 1 which shows a fragment of some event log. Only two traces are shown, both containing four events. Each event has a unique id and several properties. For example event 35654423 belongs

Table 1. A fragment of some event log

case id	event id	properties				
		timestamp	activity	resource	cost	...
x123	35654423	30-12-2011:11.02	a	John	300	...
x123	35654424	30-12-2011:11.06	b	John	400	...
x123	35654425	30-12-2011:11.12	c	John	100	...
x123	35654426	30-12-2011:11.18	d	John	400	...
x128	35655526	30-12-2011:16.10	a	Ann	300	...
x128	35655527	30-12-2011:16.14	c	John	450	...
x128	35655528	30-12-2011:16.26	b	Pete	350	...
x128	35655529	30-12-2011:16.36	d	Ann	300	...
...	...	...	...	...	...	...

to case $x123$ and is an instance of activity a that occurred on December 30th at 11.02, was executed by John, and cost 300 euros. The second trace (case $x128$) starts with event 35655526 and also refers to an instance of activity a. The information depicted in Table 1 is the typical event data that can be extracted from today's systems.

Systems store events in very different ways. Process-aware information systems (e.g., workflow management systems) provide dedicated audit trails. In other systems, this information is typically scattered over several tables. For example, in a hospital events related to a particular patient may be stored in different tables and even different systems. For many applications of process mining, one needs to extract event data from different sources, merge these data, and convert the result into a suitable format. We advocate the use of the so-called *XES (eXtensible Event Stream)* format that can be read directly by ProM ([5,57]). XES is the successor of MXML. Based on many practical experiences with MXML, the XES format has been made less restrictive and truly extendible. In September 2010, the format was adopted by the IEEE Task Force on Process Mining. The format is supported by tools such as ProM (as of version 6), Nitro, XESame, and OpenXES. See www.xes-standard.org for detailed information about the standard. XES is able to store the information shown in Table 1. Most of this information is optional, i.e., if it is there, it can be used for process mining, but it is not necessary for control-flow discovery.

In this article, we focus on control-flow discovery. Therefore, we only consider the activity column in Table 1. This means that an event is linked to a case (process instance) and an activity, and no further attributes are needed. Events are ordered (per case), but do not need to have explicit timestamps. This allows us to use the following simplified definition of an event log.

Definition 1 (Event, Trace, Event log). *Let A be a set of activities. $\sigma \in A^*$ is a* trace*, i.e., a sequence of events. $L \in \mathbb{B}(A^*)$ is an* event log*, i.e., a multi-set of traces.*

The first four events in Table 1 form a trace $\langle a, b, c, d\rangle$. This trace represents the path followed by case $x123$. The second case ($x128$) can be represented by the trace $\langle a, c, b, d\rangle$. Note that there may be multiple cases that have the same trace. Therefore, an event log is defined as a *multi-set* of traces.

A *multi-set* (also referred to as *bag*) is like a set where each element may occur multiple times. For example, $[horse, cow^5, duck^2]$ is the multi-set with eight elements: one horse, five cows and two ducks. $\mathbb{B}(X)$ is the set of multi-sets (bags) over X. We assume the usual operators on multi-sets, e.g., $X \cup Y$ is the union of X and Y, $X \setminus Y$ is the difference between X and Y, $x \in X$ tests if x appears in X, and $X \leq Y$ evaluates to true if X is contained in Y. For example, $[horse, cow^2] \cup [horse^2, duck^2] = [horse^3, cow^2, duck^2]$, $[horse^3, cow^4] \setminus [cow^2] = [horse^3, cow^2]$, $[horse, cow^2] \leq [horse^2, cow^3]$, and $[horse^3, cow^1] \not\leq [horse^2, cow^2]$. Note that sets can be considered as bags having only one instance of every element. Hence, we can mix sets and bags, e.g., $\{horse, cow\} \cup [horse^2, cow^3] = [horse^3, cow^4]$.

For practical applications of process mining it is essential to differentiate between traces that are infrequent or even unique (multiplicity of 1) and traces that are frequent. Therefore, an event log is a *multi-set of traces* rather than an ordinary set. However, in this article we focus on the foundations of process discovery thereby often abstracting from noise and frequencies. See [2] for techniques that take frequencies into account. This book also describes various case studies showing the importance of multiplicities.

In the remainder, we will use the following example log: $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$. L_1 contains information about 22 cases; five cases following trace $\langle a, b, c, d\rangle$, eight cases following trace $\langle a, c, b, d\rangle$, and nine cases following trace $\langle a, e, d\rangle$. Note that such a simple representation can be extracted from sources such as Table 1, MXML, XES, or any other format that links events to cases and activities.

3.2 Petri Nets

The goal of process discovery is to distil a process model from some event log. Here we use *Petri nets* [50] to represent such models. In fact, we extract a subclass of Petri nets known as *workflow nets* (WF-nets) [1].

Definition 2. *An* Petri net *is a tuple* (P, T, F) *where:*

1. P *is a finite set of* places*,*
2. T *is a finite set of* transitions *such that* $P \cap T = \emptyset$*, and*
3. $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs, called the* flow relation*.*

An example Petri net is shown in Figure 3. This Petri net has six places represented by circles and four transitions represented by squares. Places may contain tokens. For example, in Figure 3 both $p1$ and $p6$ contain one token, $p3$ contains two tokens, and the other places are empty. The state, also called *marking*, is the distribution of tokens over places. A *marked* Petri net is a pair (N, M), where $N = (P, T, F)$ is a Petri net and where $M \in \mathbb{B}(P)$ is a bag over P denoting the

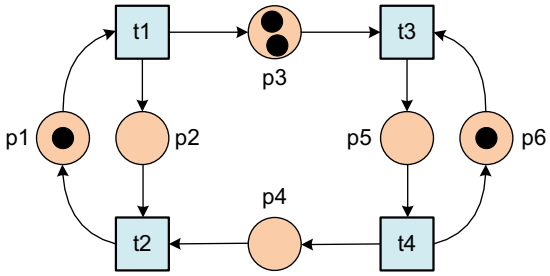


Fig. 3. A Petri net with six places ($p1$, $p2$, $p3$, $p4$, $p5$, and $p6$) and four transitions ($t1$, $t2$, $t3$, and $t4$)

marking of the net. The initial marking of the Petri net shown in Figure 3 is $[p1, p3^2, p6]$. The set of all marked Petri nets is denoted $\mathcal{N}$.

Let $N = (P, T, F)$ be a Petri net. Elements of $P \cup T$ are called *nodes*. A node x is an *input node* of another node y iff there is a directed arc from x to y (i.e., $(x, y) \in F$). Node x is an *output node* of y iff $(y, x) \in F$. For any $x \in P \cup T$, ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. In Figure 3, ${}^\bullet t3 = \{p3, p6\}$ and $t3^\bullet = \{p5\}$.

The dynamic behavior of such a marked Petri net is defined by the so-called *firing rule*. A transition is *enabled* if each of its input places contains a token. An enabled transition can *fire* thereby consuming one token from each input place and producing one token for each output place.

Definition 3 (Firing rule). *Let (N, M) be a marked Petri net with $N = (P, T, F)$. Transition $t \in T$ is* enabled*, denoted $(N, M)[t\rangle$, iff ${}^\bullet t \leq M$. The* firing rule $\_\,[\_\rangle\,\_ \subseteq \mathcal{N} \times T \times \mathcal{N}$ *is the smallest relation satisfying for any $(N, M) \in \mathcal{N}$ and any $t \in T$, $(N, M)[t\rangle \Rightarrow (N, M)\ [t\rangle\ (N, (M \setminus {}^\bullet t) \cup t^\bullet)$.*

In the marking shown in Figure 3, both $t1$ and $t3$ are enabled. The other two transitions are not enabled because at least one of the input places is empty. If $t1$ fires, one token is consumed (from $p1$) and two tokens are produced (one for $p2$ and one for $p3$). Formally, $(N, [p1, p3^2, p6])\ [t1\rangle\ (N, [p2, p3^3, p6])$. So the resulting marking is $[p2, p3^3, p6]$. If $t3$ fires in the initial state, two tokens are consumed (one from $p3$ and one from $p6$) and one token is produced (for $p5$). Formally, $(N, [p1, p3^2, p6])\ [t3\rangle\ (N, [p1, p3, p5])$.

Let (N, M_0) with $N = (P, T, F)$ be a marked P/T net. A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, M_0) iff, for some natural number $n \in \mathbb{N}$, there exist markings $M_1, \ldots, M_n$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = \langle t_1 \ldots t_n \rangle$ and, for all i with $0 \leq i < n$, $(N, M_i)[t_{i+1}\rangle$ and $(N, M_i)\ [t_{i+1}\rangle\ (N, M_{i+1})$.

Let (N, M_0) be the marked Petri net shown in Figure 3, i.e., $M_0 = [p1, p3^2, p6]$. The empty sequence $\sigma = \langle\ \rangle$ is enabled in (N, M_0). The sequence $\sigma = \langle t1, t3 \rangle$ is also enabled and results in marking $[p2, p3^2, p5]$. Another possible firing sequence is $\sigma = \langle t3, t4, t3, t1, t4, t3, t2, t1 \rangle$. A marking M is *reachable* from the initial

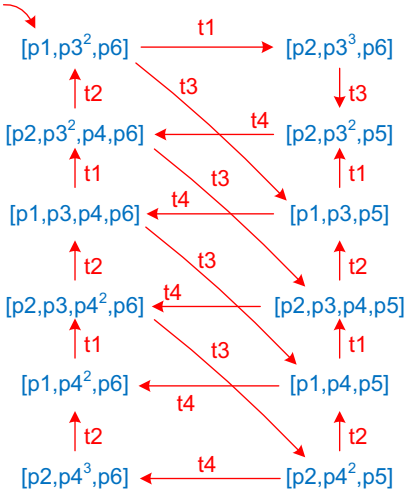


Fig. 4. The reachability graph of the marked Petri net shown in Figure 3

marking M_0 iff there exists a sequence of enabled transitions whose firing leads from M_0 to M. The set of reachable markings of (N, M_0) is denoted $[N, M_0\rangle$.

For the marked Petri net shown in Figure 3 there are 12 reachable states. These states can be computed using the so-called *reachability graph* shown in Figure 4. All nodes correspond to reachable markings and each arc corresponds to the firing of a particular transition. Any path in the reachability graph corresponds to a possible firing sequence. For example, using Figure 4 is is easy to see that $\langle t3, t4, t3, t1, t4, t3, t2, t1\rangle$ is indeed possible and results in $[p2, p3, p4, p5]$. A marked net may be unbounded, i.e., have an infinite number or reachable states. In this case, the reachability graph is infinitely large, but one can still construct the so-called coverability graph [50].

3.3 Workflow Nets

For process discovery, we look at processes that are instantiated multiple times, i.e., the same process is executed for multiple cases. For example, the process of handling insurance claims may be executed for thousands or even millions of claims. Such processes have a clear starting point and a clear ending point. Therefore, the following subclass of Petri nets (WF-nets) is most relevant for process discovery.

Definition 4 (Workflow nets). *Let $N = (P, T, F)$ be a Petri net and $\bar{t}$ a fresh identifier not in $P \cup T$. N is a* workflow net *(WF-net) iff:*

1. *object creation: P contains an input place i (also called source place) such that ${}^\bullet i = \emptyset$,*

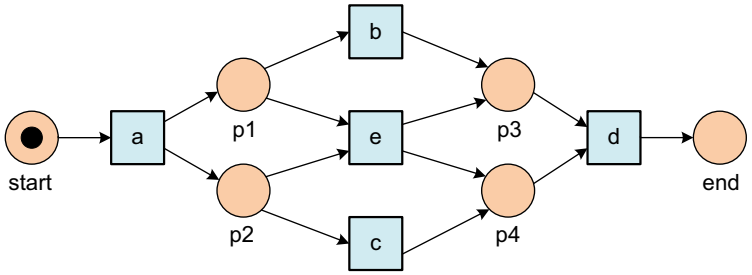


Fig. 5. A workflow net with source place $i = start$ and sink place $o = end$

2. *object completion: P contains an output place o (also called sink place) such that $o^\bullet = \emptyset$,*
3. *connectedness: $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\})$ is strongly connected, i.e., there is a directed path between any pair of nodes in $\bar{N}$.*

Clearly, Figure 3 is not a WF-net because a source and sink place are missing. Figure 5 shows an example of a WF-net: ${}^\bullet start = \emptyset$, $end^\bullet = \emptyset$, and every node is on a path from *start* to *end*.

The Petri net depicted in Figure 1 is another example of a WF-net. Not every WF-net represents a correct process. For example, a process represented by a WF-net may exhibit errors such as deadlocks, tasks which can never become active, livelocks, garbage being left in the process after termination, etc. Therefore, we define the following correctness criterion.

Definition 5 (Soundness). *Let $N = (P, T, F)$ be a WF-net with input place i and output place o. N is* sound *iff:*

1. *safeness: $(N, [i])$ is safe, i.e., places cannot hold multiple tokens at the same time,*
2. *proper completion: for any marking $M \in [N, [i]\rangle$, $o \in M$ implies $M = [o]$,*
3. *option to complete: for any marking $M \in [N, [i]\rangle$, $[o] \in [N, M\rangle$, and*
4. *absence of dead tasks: $(N, [i])$ contains no dead transitions (i.e., for any $t \in T$, there is a firing sequence enabling t).*

The WF-nets shown in figures 5 and 1 are sound. Soundness can be verified using standard Petri-net-based analysis techniques. In fact soundness corresponds to liveness and safeness of the corresponding short-circuited net [1]. This way efficient algorithms and tools can be applied. An example of a tool tailored towards the analysis of WF-nets is Woflan [55]. This functionality is also embedded in our process mining tool ProM [5].

3.4 Problem Definition and Approaches

After introducing events logs and WF-nets, we can define the main goal of process discovery.

Definition 6 (Process discovery). *Let L be an event log over A, i.e., $L \in \mathbb{B}(A^*)$. A* process discovery algorithm *is a function γ that maps any log L onto a Petri net $\gamma(L) = (N, M)$. Ideally, N is a sound WF-net and all traces in L correspond to possible firing sequences of (N, M).*

The goal is to find a process model that can "replay" all cases recorded in the log, i.e., all traces in the log are possible firing sequences of the discovered WF-net. Assume that $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$. In this case the WF-net shown in Figure 5 is a good solution. All traces in L_1 correspond to firing sequences of the WF-net and vice versa. Throughout this article, we use L_1 as an example log. Note that it may be possible that some of the firing sequences of the discovered WF-net do not appear in the log. This is acceptable as one cannot assume that all possible sequences have been observed. For example, if there is a loop, the number of possible firing sequences is infinite. Even if the model is acyclic, the number of possible sequences may be enormous due to choices and parallelism. Later in this article, we will discuss the quality of discovered models in more detail.

Since the mid-nineties several groups have been working on techniques for process mining [7, 9, 10, 25, 29, 32, 33, 58], i.e., discovering process models based on observed events. In [6] an overview is given of the early work in this domain. The idea to apply process mining in the context of workflow management systems was introduced in [10]. In parallel, Datta [29] looked at the discovery of business process models. Cook et al. investigated similar issues in the context of software engineering processes [25]. Herbst [40] was one of the first to tackle more complicated processes, e.g., processes containing duplicate tasks.

Most of the classical approaches have problems dealing with concurrency. The α-algorithm [9] is an example of a simple technique that takes concurrency as a starting point. However, this simple algorithm has problems dealing with complicated routing constructs and noise (like most of the other approaches described in literature). In [32, 33] a more robust but less precise approach is presented.

Recently, people started using the "theory of regions" to process discovery. There are two approaches: state-based regions and language-based regions. State-based regions can be used to convert a transition system into a Petri net [13, 15, 23, 24, 26, 27, 35]. Language-based regions add places as long as it is still possible to replay the log [14, 17, 19, 28, 42, 43].

More from a theoretical point of view, the process discovery problem is related to the work discussed in [12, 37, 38, 49]. In these papers the limits of inductive inference are explored. For example, in [38] it is shown that the computational problem of finding a minimum finite-state acceptor compatible with given data is NP-hard. Several of the more generic concepts discussed in these papers can be translated to the domain of process mining. It is possible to interpret the problem described in this article as an inductive inference problem specified in terms of rules, a hypothesis space, examples, and criteria for successful inference. The comparison with literature in this domain raises interesting questions for process mining, e.g., how to deal with negative examples (i.e., suppose that

besides log L there is a log L' of traces that are not possible, e.g., added by a domain expert). However, despite the relations with the work described in [12,37,38,49] there are also many differences, e.g., we are mining at the net level rather than sequential or lower level representations (e.g., Markov chains, finite state machines, or regular expressions), tackle concurrency, and do not assume negative examples or complete logs.

The above approaches assume that there is no noise or infrequent behavior. For approaches dealing with these problems we refer to the work done by Christian Günther [39], Ton Weijters [58], and Ana Karla Alves de Medeiros [47].

4 α-Algorithm

After introducing the process discovery problem and providing an overview of approaches described in literature, we focus on the α-algorithm [9]. The α-algorithm is not intended as a practical mining technique as it has problems with noise, infrequent/incomplete behavior, and complex routing constructs. Nevertheless, it provides a good introduction into the topic. The α-algorithm is very simple and many of its ideas have been embedded in more complex and robust techniques. Moreover, it was the first algorithm to really address the discovery of concurrency.

4.1 Basic Idea

The α-algorithm scans the event log for particular patterns. For example, if activity a is followed by b but b is never followed by a, then it is assumed that there is a causal dependency between a and b. To reflect this dependency, the corresponding Petri net should have a place connecting a to b. We distinguish four log-based ordering relations that aim to capture relevant patterns in the log.

Definition 7 (Log-based ordering relations). *Let L be an event log over A, i.e., $L \in \mathbb{B}(A^*)$. Let $a, b \in A$:*

- *$a >_L b$ iff there is a trace $\sigma = \langle t_1, t_2, t_3, \ldots t_n \rangle$ and $i \in \{1, \ldots, n-1\}$ such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$,*
- *$a \rightarrow_L b$ iff $a >_L b$ and $b \not>_L a$,*
- *$a \#_L b$ iff $a \not>_L b$ and $b \not>_L a$, and*
- *$a \|_L b$ iff $a >_L b$ and $b >_L a$.*

Consider for example $L_1 = [\langle a, b, c, d \rangle^5, \langle a, c, b, d \rangle^8, \langle a, e, d \rangle^9]$. $c >_{L_1} d$ because d directly follows c in trace $\langle a, b, c, d \rangle$. However, $d \not>_{L_1} c$ because c never directly follows d in any trace in the log.

$>_{L_1} = \{(a, b), (a, c), (a, e), (b, c), (c, b), (b, d), (c, d), (e, d)\}$ contains all pairs of activities in a "directly follows" relation. $c \rightarrow_{L_1} d$ because sometimes d directly follows c and never the other way around ($c >_{L_1} d$ and $d \not>_{L_1} c$). $\rightarrow_{L_1} = \{(a, b), (a, c), (a, e), (b, d), (c, d), (e, d)\}$ contains all pairs of activities in

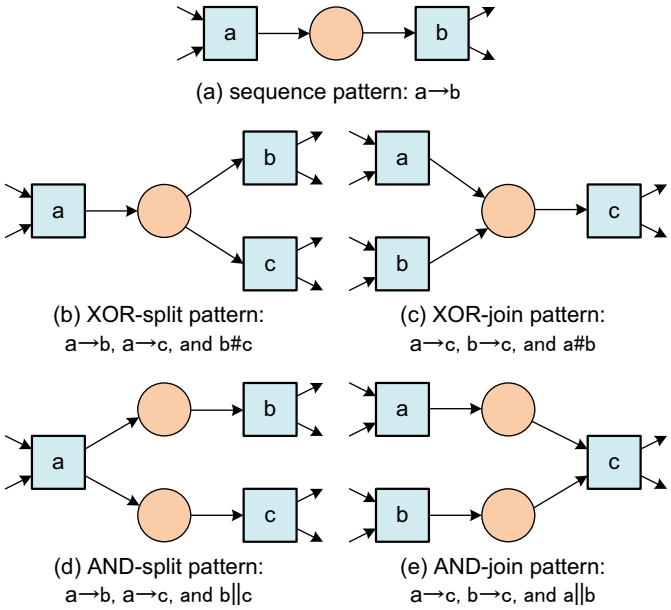


Fig. 6. Typical process patterns and the footprints they leave in the event log

a "causality" relation. $b\|_{L_1}c$ because $b >_{L_1} c$ and $c >_{L_1} b$, i.e, sometimes c follows b and sometimes the other way around. $\|_{L_1} = \{(b,c),(c,b)\}$. $b\#_{L_1}e$ because $b \not>_{L_1} e$ and $e \not>_{L_1} b$. $\#_{L_1} = \{(a,a),(a,d),(b,b),(b,e),(c,c),(c,e),(d,a),(d,d),(e,b),(e,c),(e,e)\}$. Note that for any log L over A and $x,y \in A$: $x \rightarrow_L y$, $y \rightarrow_L x$, $x\#_L y$, or $x\|_L y$.

The log-based ordering relations can be used to discover patterns in the corresponding process model as is illustrated in Figure 6. If a and b are in sequence, the log will show $a >_L b$. If after a there is a choice between b and c, the log will show $a \rightarrow_L b$, $a \rightarrow_L c$, and $b\#_L c$ because a can be followed by b and c, but b will not be followed by c and vice versa. The logical counterpart of this so-called XOR-split pattern is the XOR-join pattern as shown in Figure 6(b-c). If $a \rightarrow_L c$, $b \rightarrow_L c$, and $a\#_L b$, then this suggests that after the occurrence of either a or b, c should happen. Figure 6(d-e) shows the so-called AND-split and AND-join patterns. If $a \rightarrow_L b$, $a \rightarrow_L c$, and $b\|_L c$, then it appears that after a both b and c can be executed in parallel (AND-split pattern). If $a \rightarrow_L c$, $b \rightarrow_L c$, and $a\|_L b$, then it appears that c needs to synchronize a and b (AND-join pattern).

Figure 6 only shows simple patterns and does not present the additional conditions needed to extract the patterns. However, the figure nicely illustrates the basic idea.

Consider for example WF-net N_2 depicted in Figure 7 and the log event log $L_2 = [\langle a,b,c,d,e,f,b,d,c,e,g\rangle, \langle a,b,d,c,e,g\rangle, \langle a,b,c,d,e,f,b,c,d,e,f,b,d,c,e,g\rangle]$. The α-algorithm constructs WF-net N_2 based on L_2. Note that the

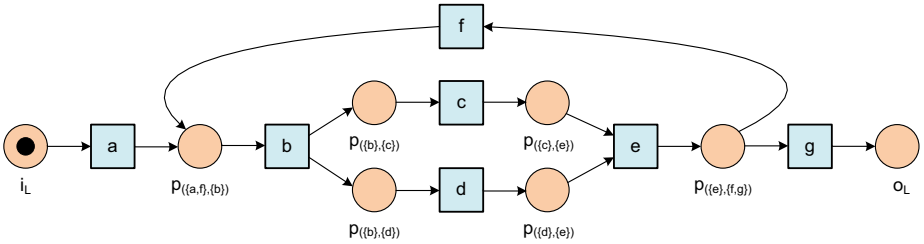


Fig. 7. WF-net N_2 derived from $L_2 = [\langle a, b, c, d, e, f, b, d, c, e, g\rangle, \langle a, b, d, c, e, g\rangle, \langle a, b, c, d, e, f, b, c, d, e, f, b, d, c, e, g\rangle]$

patterns in the model indeed match the log-based ordering relations extracted from the event log. Consider for example the process fragment involving b, c, d, and e. Obviously, this fragment can be constructed based on $b \rightarrow_{L_2} c$, $b \rightarrow_{L_2} d$, $c \| _{L_2} d$, $c \rightarrow_{L_2} e$, and $d \rightarrow_{L_2} e$. The choice following e is revealed by $e \rightarrow_{L_2} f$, $e \rightarrow_{L_2} g$, and $f \#_{L_2} g$. Etc.

Another example is shown in Figure 8. WF-net N_3 can be derived from $L_3 = [\langle a, c, d\rangle^{45}, \langle b, c, d\rangle^{42}, \langle a, c, e\rangle^{38}, \langle b, c, e\rangle^{22}]$. Note that here there are two start and two end activities. These can be found easily by looking for the first and last activities in traces.

4.2 Algorithm

After showing the basic idea and some examples, we describe the α-algorithm.

Definition 8 (α-algorithm). *Let L be an event log over T. $\alpha(L)$ is defined as follows.*

1. $T_L = \{t \in T \mid \exists_{\sigma \in L}\ t \in \sigma\}$,
2. $T_I = \{t \in T \mid \exists_{\sigma \in L}\ t = first(\sigma)\}$,
3. $T_O = \{t \in T \mid \exists_{\sigma \in L}\ t = last(\sigma)\}$,
4. $X_L = \{(A, B) \mid A \subseteq T_L \ \wedge\ A \neq \emptyset \ \wedge\ B \subseteq T_L \ \wedge\ B \neq \emptyset \ \wedge\ \forall_{a \in A} \forall_{b \in B}\ a \rightarrow_L b \ \wedge\ \forall_{a_1, a_2 \in A}\ a_1 \#_L a_2 \ \wedge\ \forall_{b_1, b_2 \in B}\ b_1 \#_L b_2\}$,
5. $Y_L = \{(A, B) \in X_L \mid \forall_{(A', B') \in X_L} A \subseteq A' \ \wedge B \subseteq B' \Longrightarrow (A, B) = (A', B')\}$,

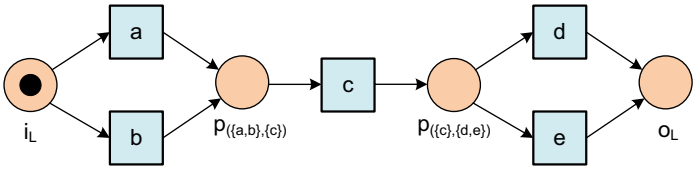


Fig. 8. WF-net N_3 derived from $L_3 = [\langle a, c, d\rangle^{45}, \langle b, c, d\rangle^{42}, \langle a, c, e\rangle^{38}, \langle b, c, e\rangle^{22}]$

6. $P_L = \{p_{(A,B)} \mid (A,B) \in Y_L\} \cup \{i_L, o_L\}$,
7. $F_L = \{(a, p_{(A,B)}) \mid (A,B) \in Y_L \wedge a \in A\} \cup \{(p_{(A,B)}, b) \mid (A,B) \in Y_L \wedge b \in B\} \cup \{(i_L, t) \mid t \in T_I\} \cup \{(t, o_L) \mid t \in T_O\}$, *and*
8. $\alpha(L) = (P_L, T_L, F_L)$.

L is an event log over some set T of activities. In Step 1 it is checked which activities do appear in the log (T_L). These will correspond to the transitions of the generated WF-net. T_I is the set of start activities, i.e., all activities that appear first in some trace (Step 2). T_O is the set of end activities, i.e., all activities that appear last in some trace (Step 3). Steps 4 and 5 form the core of the α-algorithm. The challenge is to find the places of the WF-net and their connections. We aim at constructing places named $p_{(A,B)}$ such that A is the set of input transitions (${}^\bullet p_{(A,B)} = A$) and B is the set of output transitions ($p_{(A,B)}{}^\bullet = B$).

The basic idea for finding $p_{(A,B)}$ is shown in Figure 9. All elements of A should have causal dependencies with all elements of B, i.e., for any $(a,b) \in A \times B$: $a \rightarrow_L b$. Moreover, the elements of A should never follow any of the other elements, i.e., for any $a_1, a_2 \in A$: $a_1 \#_L a_2$. A similar requirement holds for B.

Let us consider $L_1 = [\langle a,b,c,d\rangle^5, \langle a,c,b,d\rangle^8, \langle a,e,d\rangle^9]$. Clearly $A = \{a\}$ and $B = \{b,e\}$ meet the requirements stated in Step 4. Also note that $A' = \{a\}$ and $B' = \{b\}$ meet the same requirements. X_L is the set of all such pairs that meet the requirements just mentioned. In this case, $X_{L_1} = \{(\{a\},\{b\}), (\{a\},\{c\}), (\{a\},\{e\}), (\{a\},\{b,e\}), (\{a\},\{c,e\}), (\{b\},\{d\}), (\{c\},\{d\}), (\{e\},\{d\}), (\{b,e\},\{d\}), (\{c,e\},\{d\})\}$. If one would insert a place for any element in X_{L_1} there would be too many places. Therefore, only the "maximal pairs" (A,B) should be included. Note that for any pair $(A,B) \in X_L$, non-empty set $A' \subseteq A$, and non-empty set $B' \subseteq B$, it is implied that $(A',B') \in X_L$. In Step 5 all non-maximal pairs are removed. So $Y_{L_1} = \{(\{a\},\{b,e\}), (\{a\},\{c,e\}), (\{b,e\},\{d\}), (\{c,e\},\{d\})\}$.

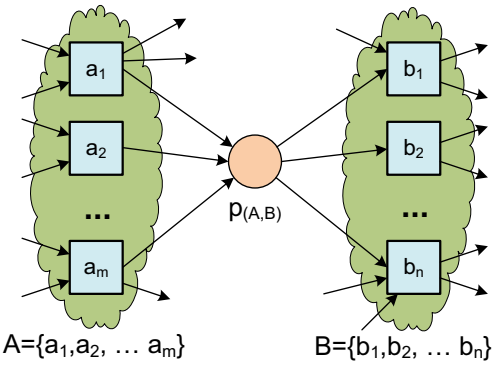


Fig. 9. Place $p_{(A,B)}$ connects the transitions in set A to the transitions in set B

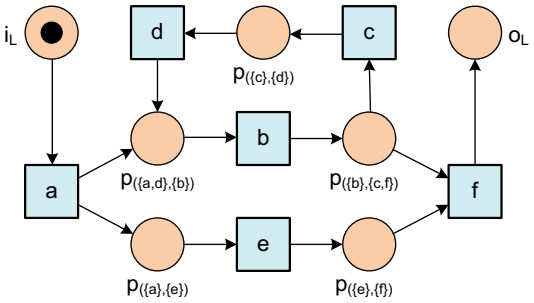


Fig. 10. WF-net N_4 derived from $L_4 = [\langle a,b,e,f\rangle^2, \langle a,b,e,c,d,b,f\rangle^3, \langle a,b,c,e,d,b,f\rangle^2, \langle a,b,c,d,e,b,f\rangle^4, \langle a,e,b,c,d,b,f\rangle^3]$

Every element of $(A,B) \in Y_L$ corresponds to a place $p_{(A,B)}$ connecting transitions A to transitions B. In addition P_L also contains a unique source place i_L and a unique sink place o_L (cf. Step 6).

In Step 7 the arcs are generated. All start transitions in T_I have i_L as an input place and all end transitions T_O have o_L as output place. All places $p_{(A,B)}$ have A as input nodes and B as output nodes. The result is a Petri net $\alpha(L) = (P_L, T_L, F_L)$ that describes the behavior seen in event log L.

Thus far we presented three logs and three WF-nets. Clearly $\alpha(L_2) = N_2$, and $\alpha(L_3) = N_3$. In figures 7 and 8 the places are named based on the sets Y_{L_2} and Y_{L_3}. Moreover, $\alpha(L_1) = N_1$ modulo renaming of places (because different place names are used in Figure 5). These examples show that the α-algorithm is indeed able to discover WF-nets based event logs.

Figure 10 shows another example. WF-net N_4 can be derived from $L_4 = [\langle a,b,e,f\rangle^2, \langle a,b,e,c,d,b,f\rangle^3, \langle a,b,c,e,d,b,f\rangle^2, \langle a,b,c,d,e,b,f\rangle^4, \langle a,e,b,c,d,b,f\rangle^3]$, i.e., $\alpha(L_4) = N_4$.

The WF-net in Figure 1 is discovered when applying the α-algorithm to the event log in the same figure.

4.3 Limitations

In [9] it was shown that the α-algorithm is able to discover a large class of WF-nets if one assumes that the log is *complete* with respect to the log-based ordering relation $>_L$. This assumption implies that, for any event log L, $a >_L b$ if a can be directly followed by b. We revisit the notion of completeness later in this article.

Even if we assume that the log is complete, the α-algorithm has some problems. There are many different WF-nets that have the same possible behavior, i.e., two models can be structurally different but trace equivalent. Consider for example $L_5 = [\langle a,c,e,g\rangle^2, \langle a,e,c,g\rangle^3, \langle b,d,f,g\rangle^2, \langle b,f,d,g\rangle^4]$. $\alpha(L_5)$ is shown in Figure 11. Although the model is able to generate the observed behavior, the resulting WF-net is needlessly complex. Two of the input places of g are redundant, i.e., they can be removed without changing the behavior.

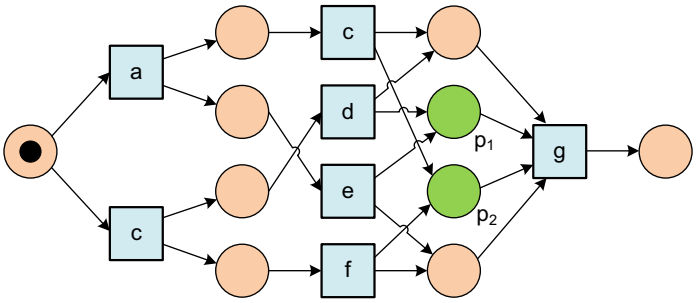


Fig. 11. WF-net N_5 derived from $L_5 = [\langle a, c, e, g\rangle^2, \langle a, e, c, g\rangle^3, \langle b, d, f, g\rangle^2, \langle b, f, d, g\rangle^4]$

The places denoted as p_1 and p_2 are so-called implicit places and can be removed without allowing for more traces. In fact, Figure 11 shows only one of many possible trace equivalent WF-nets.

The original α-algorithm has problems dealing with short loops, i.e., loops of length 1 or 2. This is illustrated by WF-net N_6 in Figure 12 which shows the result of applying the basic algorithm to $L_6 = [\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2]$. It is easy to see that the model does not allow for $\langle a, c\rangle$ and $\langle a, b, b, c\rangle$. In fact, in N_6, transition b needs to be executed precisely once and there is an implicit place connecting a and c. This problem can be addressed easily as shown in [46]. Using an improved version of the α-algorithm one can discover the WF-net shown in Figure 13.

A more difficult problem is the discovery of so-called non-local dependencies resulting from non-free choice process constructs. An example is shown in Figure 14. This net would be a good candidate after observing for example

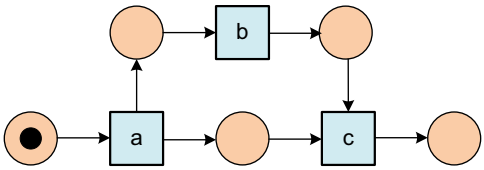


Fig. 12. Incorrect WF-net N_6 derived from $L_6 = [\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2]$

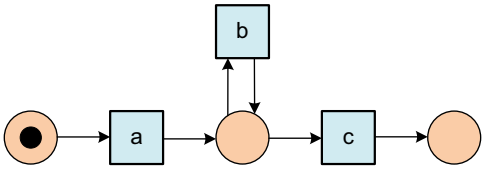


Fig. 13. WF-net N_7 having a so-called "short-loop"

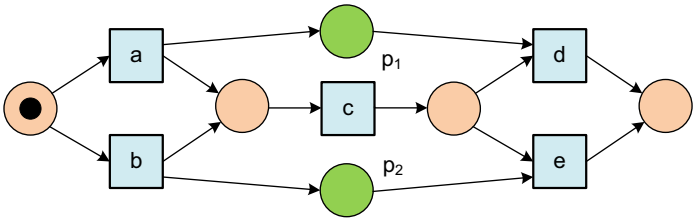


Fig. 14. WF-net N_8 having a non-local dependency

$L_8 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$. However, the α-algorithm will derive the WF-net without the place labeled p_1 and p_2. Hence, $\alpha(L_8) = N_3$ shown in Figure 8 although the traces $\langle a, c, e\rangle$ and $\langle b, c, d\rangle$ do not appear in L_8. Such problems can be (partially) resolved using refined versions of the α-algorithm such as the one presented in [59].

The above examples show that the α-algorithm is able to discover a large class of models. The basic 8-line algorithm has some limitations when it comes to particular process patterns (e.g., short-loops and non-local dependencies). Some of these problems can be solved using various refinements. However, several more fundamental problems remain as shown next.

5 Challenges

The α-algorithm was one of the first process discovery algorithms to adequately capture concurrency. Today there are much better algorithms that overcome the weaknesses of the α-algorithm. These are either variants of the α-algorithm or algorithms that use a completely different approach, e.g., genetic mining or synthesis based on regions [34]. Later we will describe some of these approaches. However, first we discuss the main requirements for a good process discovery algorithm.

To discover a suitable process model it is assumed that the event log contains a *representative sample of behavior*. There are two related phenomena that may make an event log less representative for the process being studied:

- *Noise:* the event log contains rare and infrequent behavior not representative for the typical behavior of the process.
- *Incompleteness:* the event log contains too few events to be able to discover some of the underlying control-flow structures.

Often we would like to abstract from noise when discovering a process. This does not mean that noise is not relevant. In fact, the goal of conformance checking is to identify exceptions and deviations. However, for process discovery it makes no sense to include noisy behavior in the model as this will clutter the diagram and has little predictive value. Whereas noise refers to the problem of having "too much data" (describing rare behavior), completeness refers to the problem of having "too little data". To illustrate the relevance of completeness,

consider a process consisting of 10 activities that can be executed in parallel and a corresponding log that contains information about 10,000 cases. The total number of possible interleavings in the model with 10 concurrent activities is $10! = 3{,}628{,}800$. Hence, it is impossible that each interleaving is present in the log as there are fewer cases (10,000) than potential traces (3,628,800). Even if there are 3,628,800 cases in the log, it is extremely unlikely that all possible variations are present. For the process in which 10 activities can be executed in parallel, a local notion of completeness can reduce the required number of observations dramatically. For example, for the α-algorithm only $10 \times (10 - 1) = 90$ rather than 3,628,800 different observations are needed to construct the model.

Completeness and noise refer to qualities of the event log and do not say much about the quality of the discovered model. Determining the quality of a process mining result is difficult and is characterized by many dimensions. As shown in Figure 15, we identify four main quality dimensions: *fitness*, *simplicity*, *precision*, and *generalization* [2, 4, 51].

A model with good *fitness* allows for the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end. There are various ways of defining fitness. It can be defined at the case level, e.g., the fraction of traces in the log that can be fully replayed. It can also be defined at the event level, e.g., the fraction of events in the log that are indeed possible according to the model [2, 4, 51]. Note that we defined an event log to be a multi-set of traces rather than an ordinary set: the frequencies of traces are important for determining fitness. If a trace cannot be replayed by the model, then the significance of this problem depends on the relative frequency.

The *simplicity* dimension in Figure 15 refers to *Occam's Razor*, the principle that states that "one should not increase, beyond what is necessary, the number of entities required to explain anything". Following this principle, we look for the "simplest process model" that can explain what is observed in the event

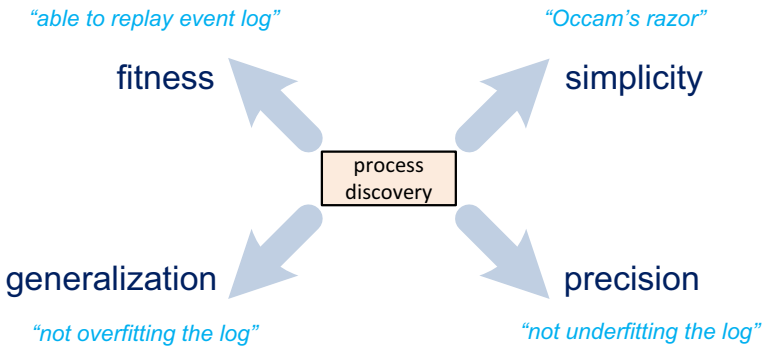


Fig. 15. Balancing the four quality dimensions: *fitness*, *simplicity*, *precision*, and *generalization* [2]

log. The complexity of the model could be defined by the number of nodes and arcs in the underlying graph. Also more sophisticated metrics can be used, e.g., metrics that take the "structuredness" or "entropy" of the model into account.

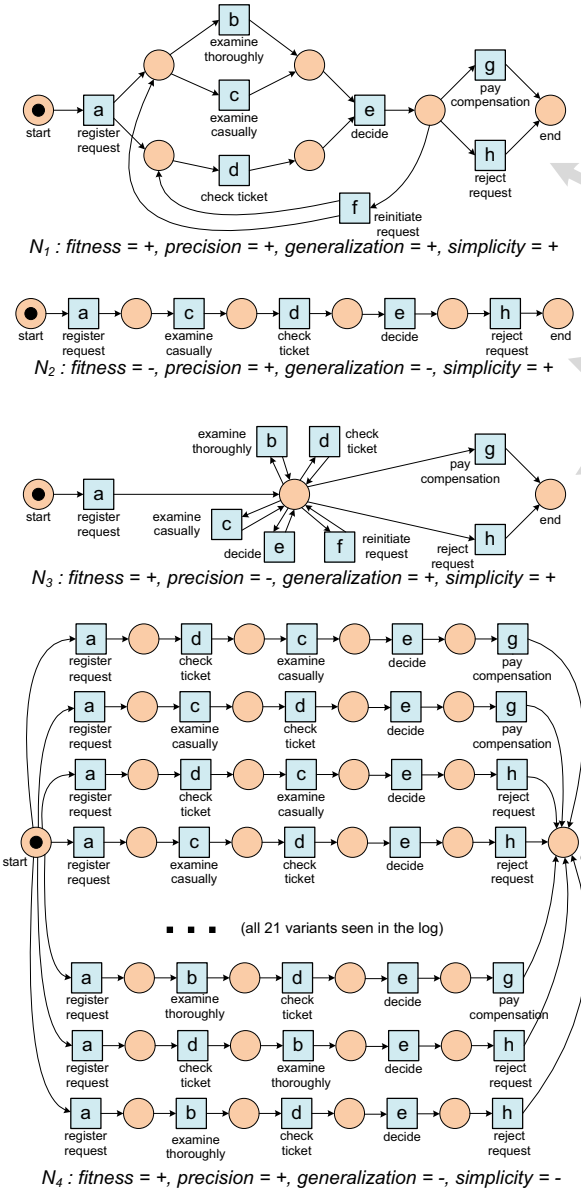
Fitness and simplicity are obvious criteria. However, this is not sufficient as will be illustrated using Figure 16. Assume that the four models that are shown are discovered based on the event log also depicted in the figure. (Note that this event log was already shown in Section 1.) There are 1391 cases. Of these 1391 cases, 455 followed the trace $\langle a, c, d, e, h\rangle$. The second most frequent trace is $\langle a, b, d, e, g\rangle$ which was followed by 191 cases.

If we apply the α-algorithm to this event log, we obtain model N_1 shown in Figure 16. A comparison of the WF-net N_1 and the log shows that this model is quite good; it is simple and has a good fitness. WF-net N_2 models only the most frequent trace, i.e., it only allows for the sequence $\langle a, c, d, e, h\rangle$. Hence, none of the other $1391 - 455 = 936$ cases fits. WF-net N_2 is simple but has a poor fitness.

Let us now consider WF-net N_3, this is a variant of the so-called "flower model" [2, 51], i.e., a model that allows for all known activities at any point in time. Note that a Petri net without any places can replay any log and has a behavior similar to the "flower model" (but is not a WF-net). Figure 16 does not show a pure "flower model", but still allows for a diversity of behaviors. N_3 captures the start and end activities well. However, the model does not put any constraints on the other activities. For example trace $\langle a, b, b, b, b, b, b, f, f, f, f, f, g\rangle$ is possible, whereas it seems unlikely that this trace is possible when looking at the event log, i.e., the behavior is very different from any of the traces in the log.

Extreme models such as the "flower model" (anything is possible) show the need for an additional dimension: *precision*. A model is precise if it does not allow for "too much" behavior. Clearly, the "flower model" lacks precision. A model that is not precise is "underfitting". Underfitting is the problem that the model over-generalizes the example behavior in the log, i.e., the model allows for behaviors very different from what was seen in the log.

WF-net N_4 in Figure 16 reveals another potential problem. This model simply enumerates the 21 different traces seen in the event log. Note that N_4 is a so-called *labeled* Petri net, i.e., multiple transitions can have the same label (there are 21 transition with label a). The WF-net in Figure 16 is precise and has a good fitness. However, N_4 is also overly complex and is "overfitting". WF-net N_4 illustrates the need to *generalize*; one should not restrict behavior to the traces seen in the log as these are just examples. Overfitting is the problem that a very specific model is generated whereas it is obvious that the log only holds example behavior, i.e., the model explains the particular sample log, but a next sample log of the same process may produce a completely different process model. Recall that logs are typically far from complete. Moreover, generalization can be used to simplify models. WF-net N_1 shown in Figure 16 allows for behavior not seen in the log, e.g., $\langle a, d, c, e, f, d, b, e, f, c, d, e, h\rangle$. Any WF-net that restricts the behavior to only seen cases will be much more complex and exclude behavior which seems likely based on similar traces in the event log.



#	trace
455	acdeh
191	abdeg
177	adceh
144	abdeh
111	acdeg
82	adceg
56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg
1391	

Fig. 16. Different Petri nets discovered for an event log containing 1391 cases

For real-life event logs it is challenging to balance the four quality dimensions shown in Figure 15. For instance, an oversimplified model is likely to have a low fitness or lack of precision. Moreover, there is an obvious trade-off between underfitting and overfitting [2, 4, 48, 51].

6 Process Discovery and the Theory of Regions

Problems similar to process discovery arise in other areas ranging from hardware design and to controller synthesis of manufacturing systems. Often the so called *theory of regions* is used to construct a Petri net from a behavioral specification (e.g., a language or a state space), such that the behavior of this net corresponds to the specified behavior (if such a net exists). The general question answered by the theory of regions is: *Given the specified behavior of a system, what is the Petri net that represents this behavior?*.

Two main types of region theory can be distinguished, namely *state-based region theory* and *language-based region theory*. The state-based theory of regions focusses on the synthesis of Petri nets from state-based models, where the state space of the Petri net is bisimilar to the given state-based model. The language-based region theory, considers a language over a finite alphabet as a behavioral specification. Using the notion of regions, a Petri net is constructed, such that all words in the language are firing sequences in that Petri net.

The aim of the theory of regions is to synthesize a precise model, with minimal generalization, while keeping a maximal fitness. The classical approaches described in this section (i.e., conventional *state-based* region theory and *language based* region theory) do not put much emphasis on simplicity. Unlike algorithms such as the heuristic miner [58], the genetic miner [47], and the fuzzy miner [39], conventional region-based methods do not compromise on precision in favor of simplicity or generalization.

In the remainder of this section, we introduce the main region theory concepts and discuss the differences between synthesis and process discovery. In Section 7 and Section 8 we show how region theory can be used in the context of process discovery.

6.1 State Based Region Theory

The *state-based region theory* [13, 15, 23, 24, 26, 27, 35] uses a transition system as input, i.e., it attempts to construct a Petri net that is bisimilar to the transition system. Hence both are behaviorally equivalent and if the system exhibits concurrency, the Petri net may be much smaller than the transition system.

Definition 9 (Transition system). *$TS = (S, E, T)$ defines a* labeled transition system *where S is the set of states, A is the set of visible activities (i.e., activities recorded in event log), $\tau \notin A$ is used to represent silent steps (i.e., actions not recorded in event log), $E = A \cup \{\tau\}$ is the set of transition labels, and $T \subseteq S \times E \times S$ is the transition relation. We use $s_1 \xrightarrow{e} s_2$*

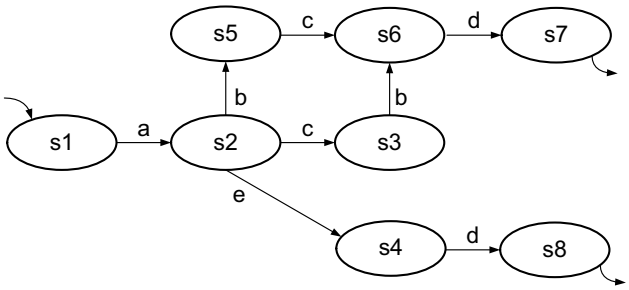


Fig. 17. A transition system with 8 states, 5 labels, 1 initial state and 2 final states

to denote a transition from state s_1 to s_2 labeled with e. Furthermore, we say that $S^s = \{s \in S \mid \nexists_{s' \in S, e \in E}\ s' \xrightarrow{e} s\} \subseteq S$ is the set of initial states, and $S^e = \{s \in S \mid \nexists_{s' \in S, e \in E}\ s \xrightarrow{e} s'\} \subseteq S$ is the set of final states.

In the transition system, a *region* corresponds to a set of states such that all states have similarly labeled input and output edges. Figure 17 shows an example of a transition system. In fact, this figure depicts the reachability graph of the Petri net in Figure 5, where the states are anonymous, i.e., they do not contain information about how many tokens are in a place.

Definition 10 (State region). *Let $TS = (S, E, T)$ be a transition system and $S' \subseteq S$ a set of states. We say S' is a region, if and only if for all $e \in E$ one of the following conditions holds:*

1. *all the transitions $s_1 \xrightarrow{e} s_2$* enter *S', i.e., $s_1 \notin S'$ and $s_2 \in S'$,*
2. *all the transitions $s_1 \xrightarrow{e} s_2$* exit *S', i.e., $s_1 \in S'$ and $s_2 \notin S'$,*
3. *all the transitions $s_1 \xrightarrow{e} s_2$* do not cross *S', i.e., $s_1, s_2 \in S'$ or $s_1, s_2 \notin S'$*

Any transition system $TS = (S, E, T)$ has two trivial regions: $\emptyset$ (the empty region) and S (the region consisting of all states). Typically, only non-trivial regions are considered. A region r' is said to be a *subregion* of another region r if $r' \subset r$. A region r is *minimal* if there is no other region r' which is a subregion of r. Region r is a *preregion* of e if there is a transition labeled with e which exits r. Region r is a *postregion* of e if there is a transition labeled with e which enters r.

For Petri net synthesis, a region corresponds to a Petri net *place* and an event corresponds to a Petri net *transition*. Thus, the main idea of the *synthesis algorithm* is the following: for each event e in the transition system, a transition labeled with e is generated in the Petri net. For each minimal region r_i a place p_i is generated. The flow relation of the Petri net is built the following way: $e \in p_i{}^\bullet$ if r_i is a preregion of e and $e \in {}^\bullet p_i$ if r_i is a postregion of e. Figure 18 shows the minimal regions of the transition system of Figure 17 and the corresponding Petri net.

The first publications on the theory of regions only dealt with a special class of transition systems called *elementary transition systems*. See [13, 15, 30] for

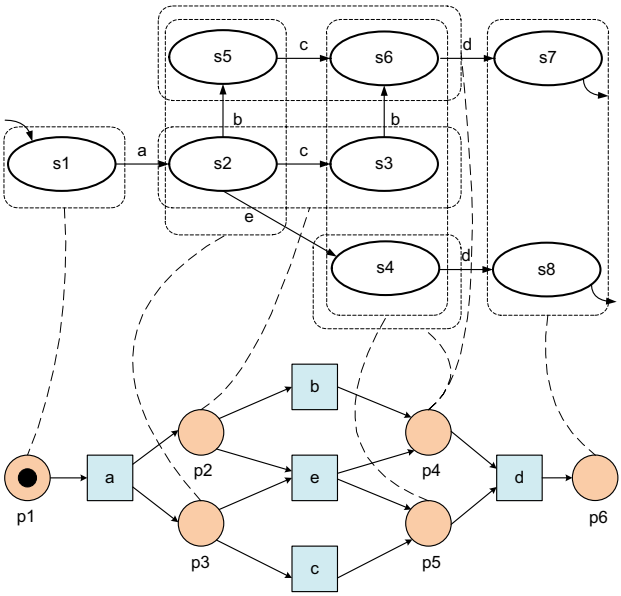


Fig. 18. The transition system of Figure 17 is converted into a Petri net using the "state regions". The six regions correspond to places in the Petri net.

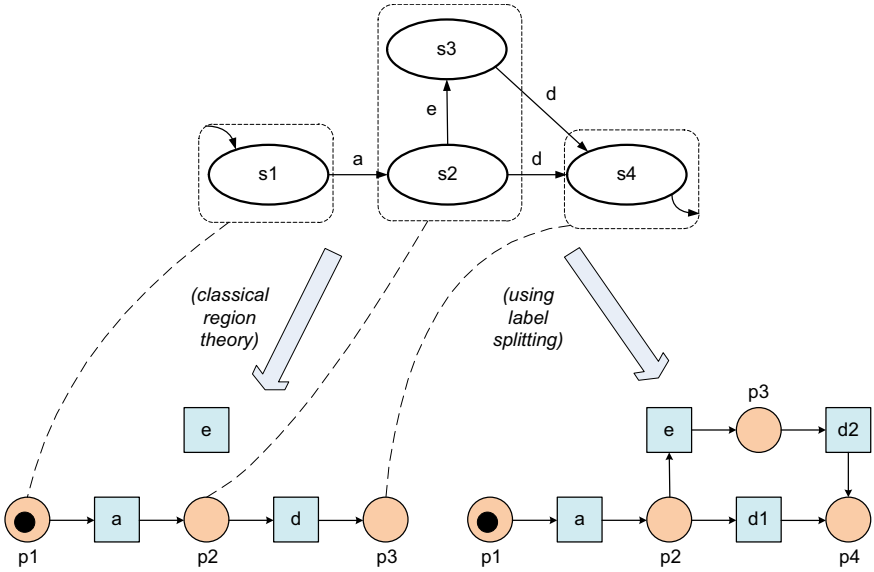


Fig. 19. The transition system is not elementary. Therefore, the generated Petri net using classical region theory is not equivalent (modulo bisimilarity). However, using "label-splitting" an equivalent Petri net can be obtained.

details. The class of elementary transition systems is very restricted. In practice, most of the time, people need to deal with arbitrary transition systems that only by coincidence fall into the class of elementary transition systems. In the papers of Cortadella et al. [26, 27], a method for handling arbitrary transition systems was presented. This approach uses *labeled Petri nets*, i.e., different transitions can refer to the same event. (WF-net N_4 in Figure 16 is an example of a labeled Petri net, e.g., there are 21 transitions labeled a.) For this approach it has been shown that the behavior (cf. reachability graph) of the synthesized Petri net is *bisimilar* to the initial transition system even if the transition system is non-elementary. More recently, in [23,24], an approach was presented where the constructed Petri net is not necessarily safe, but bounded[1]. Again, the reachability graph of the synthesized Petri net is bisimilar to the given transition system.

To illustrate the problem of non-elementary transition systems, consider Figure 19. This transition system is not elementary. The problem is that there are two states $s2$ and $s3$ that are identical in terms of regions, i.e., there is no region such that one is part of it and the other is not. As a result, the constructed Petri net on the left hand side of Figure 19 fails to construct a bisimilar Petri net. However, using label-splitting as presented in [26, 27], the Petri net on the right hand side can be obtained. This Petri net has two transitions $d1$ and $d2$ corresponding to activity d in the log. The splitting is based on the so-called notions of *excitation* and *generalized excitation region*, see [26]. As shown in [26, 27] it is always possible to construct an equivalent Petri net. However, label-splitting may lead to larger Petri nets. In [21] the authors show how to obtain the most precise model when label splitting is not allowed.

In state-based region theory, the aim is to construct a Petri net, such that its behavior is bisimilar to the given transition system. In process discovery however, we have a log as input, i.e., we have information about sequences of transitions, but not about states. In Section 7, we show how we can identify state information from event logs and then use state-based region theory for process discovery. However, we first introduce language-based region theory.

6.2 Language Based Region Theory

In addition to state-based region theory, we also consider language-based region theory [14, 17, 19, 28, 42, 43]. In their survey paper [45], Mauser and Lorenz show how for different classes of languages (step languages, regular languages and (infinite) partial languages) a Petri net can be derived such that the resulting net is the Petri net with the smallest behavior in which the words in the language are possible firing sequences.

Given a prefix-closed language $\mathcal{A}$ over some non-empty, finite set of activities A, the language-based theory of regions tries to find a finite Petri net $N(\mathcal{A})$ in which the transitions correspond to the elements in the set A and of which all sequences in the language are firing sequences (fitness criterion). Furthermore,

[1] A Petri net is safe if there can never be more than 1 token in any place. Boundedness implies that there exists an upper bound for the number of tokens in any place.

the Petri net should minimize the number of firing sequences not in the language (precision criterion).

The Petri net $N(\mathcal{A}) = (\emptyset, A, \emptyset)$ is a finite Petri net with infinitely many firing sequences allowing for any sequence involving activities A. Such a model is typically underfitting, i.e., allowing for more behavior than suggested by the event log. Therefore, the behavior of this Petri net needs to be reduced so that the Petri net still allows to reproduce all sequences in the language, but does not allow for behavior unrelated to the examples seen in the event log. This is achieved by adding places to the Petri net. The theory of regions provides a method to identify these places, using *language regions*.

Definition 11 (Language Region). *Let A be a set of activities. A region of a prefix-closed language $\mathcal{L}$ over A is a triple $(\vec{x}, \vec{y}, c)$ with $\vec{x}, \vec{y} \in \{0,1\}^A$ and $c \in \{0,1\}$, such that for each non-empty sequence $w = w' \circ a \in \mathcal{L}$, $w' \in \mathcal{L}$, $a \in A$:*

$$c + \sum_{t \in A} \left(\vec{w'}(t) \cdot \vec{x}(t) - \vec{w}(t) \cdot \vec{y}(t) \right) \geq 0$$

This can be rewritten into the inequation system:

$$c \cdot \vec{1} + M' \cdot \vec{x} - M \cdot \vec{y} \geq \vec{0}$$

where M and M' are two $|\mathcal{L}| \times |A|$ matrices with $M(w,t) = \vec{w}(t)$, and $M'(w,t) = \vec{w'}(t)$, with $w = w' \circ a$. The set of all regions of a language is denoted by $\Re(\mathcal{L})$ and the region $(\vec{0}, \vec{0}, 0)$ is called the trivial region.[2]

Consider a region $r = (\vec{x}, \vec{y}, c)$ corresponding to some place p_r. For any prefix $w = w' \circ a$ in $\mathcal{L}$, region r satisfies $c + \sum_{t \in A} \left(\vec{w'}(t) \cdot \vec{x}(t) - \vec{w}(t) \cdot \vec{y}(t) \right) \geq 0$ where c is the initial number of tokens in place p_r, $\sum_{t \in A} \vec{w'}(t) \cdot \vec{x}(t)$ is the number of tokens produced for place p_r just before firing a (note that w' is the prefix without including the last a), and $\sum_{t \in A} \vec{w}(t) \cdot \vec{y}(t)$ is the number of tokens consumed from place p_r after firing a (w is the concatenation of w' and a). $\vec{w}$ is the Parikh vector of w, i.e., $\vec{w}(t)$ is the number of times t appears in sequence w. $\vec{x}(t)$ is the number of tokens t produces for place p_r. Transition t consumes $\vec{y}(t)$ tokens from place p_r per firing. So, $\vec{w}(t) \cdot \vec{y}(t)$ is the total number of tokens t consumes from place p_r when executing w.

Figure 20 illustrates the language-based region concept using for a language over four activities ($|A|$ =4), i.e., each solution $(\vec{x}, \vec{y}, c)$ of the inequation system can be regarded in the context of a Petri net, where the region corresponds to a feasible place with preset $\{t \mid t \in T, \vec{x}(t) \geq 1\}$ and postset $\{t \mid t \in T, \vec{y}(t) \geq 1\}$, and initially marked with c tokens. In this paper, we assume arc-weights to be 0 or 1 as we aim at understandable models (i.e., $\vec{x}, \vec{y} \in \{0,1\}^A$). As shown in [14,16,28,43] it is possible to generalize the above notions to arbitrary arc-weights.

[2] To reduce calculation time, the inequation system can be rewritten to the form $[\vec{1}; M'; -M] \cdot \vec{r} \geq \vec{0}$ which can be simplified by eliminating duplicate rows.

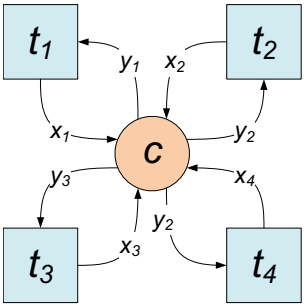


Fig. 20. Region for a language with four letters: t_1, t_2, t_3, and t_4

A place represented by a region can be added to a Petri net, without limiting its behavior with respect to traces seen in the event log. Therefore, we call such a place *feasible*.

Definition 12 (Feasible place). *Let $\mathcal{L}$ be a prefix-closed language over A and let $N = ((P, T, F), M)$ be a marked Petri net with $T = A$ and $M \in \mathbb{B}(P)$. A place $p \in P$ is called* feasible *if and only if there exists a* corresponding *region $(\vec{x}, \vec{y}, c) \in \Re(\mathcal{L})$ such that $M(p) = c$, and $\vec{x}(t) = 1$ if and only if $t \in {}^\bullet p$, and $\vec{y}(t) = 1$ if and only if $t \in p^\bullet$.*

In [16,43] it was shown that any solution of the inequation system of Definition 11 can be added to a Petri net without influencing the ability of that Petri net to replay the log. However, since there are infinitely many solutions of that inequation system (assuming arc weights), there are infinite many feasible places and the authors of [16,43] present two ways of finitely representing these places, namely a *basis representation* [43] and a *separating representation* [16,43].

6.3 Process Discovery vs. Region Theory

When comparing region theory—state-based or language based—with process discovery, some important differences should be noted. First of all, in region theory, the starting point is a *full behavioral specification*, either in the form of a (possibly infinite) transition system, or a (possibly infinite) language. Hence, the underlying assumption is that the input is *complete* and *noise free* and therefore *maximal fitness* is assured.

Second, the aim of region theory is to provide a *compact, exact representation* of that behavior in the form of a Petri net. If the net allows for more behavior than specified, then this additional behavior can be proven to be minimal, hence region theory provides *precise* results.

Finally, when region theory is directly applied in the context of process discovery [16,21,53], the resulting Petri nets typically perform poorly with respect to two of the four dimensions shown in Figure 15. The resulting models are typically overfitting (lack of generalization) and are too difficult to comprehend

(simplicity criterion). Therefore, in sections 7 and 8, we show how region theory can be modified for process discovery. The key idea is to allow the algorithms to generalize and relax on preciseness, with the aim of obtaining simpler models.

7 Process Discovery Using State-Based Region Theory

In Section 2 we introduced the concept of control-flow discovery and discussed the problems of existing approaches. In Section 6, we introduced region theory and showed the main differences with control flow discovery. In this section, we introduce a two-step approach to combine process discovery with state-based region theory [8]. In the remainder, we elaborate on these two steps and discuss challenges.

7.1 From Event Logs to Transition Systems

In the first step, we construct a transition system from the log, where we generalize from the observed behavior. Furthermore, we "massage" the output, such that the region theory used in the second step is more likely to produce a simple model. In the second step, we use classical state-based region theory to obtain a Petri net. This section describes the first and most important step. Depending on the desired properties of the model and the characteristics of the log, the algorithm can be tuned to provide a more simple and/or generic model.

The most important aspect of process discovery is *deducing the states of the operational process in the log*. Most mining algorithms have an implicit notion of state, i.e., activities are glued together in some process modeling language based on an analysis of the log and the resulting model has a behavior that can be represented as a transition system. In this section, we propose to *define states explicitly* and start with the definition of a transition system.

In some cases, the state can be derived directly, e.g., each event encodes the complete state by providing values for all relevant data attributes. However, in

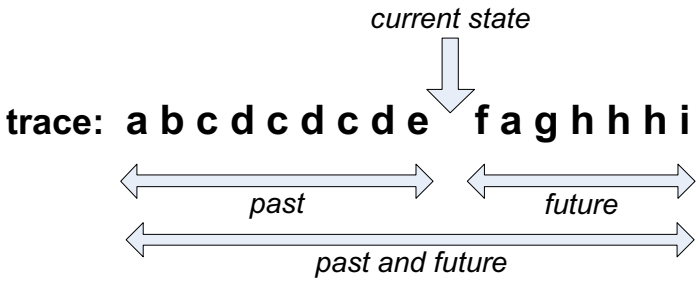


Fig. 21. Three basic "ingredients" can be considered as a basis for calculating the "process state": (1) past, (2) future, and (3) past and future

the event log we typically only see activities and not states. Hence, we need to deduce the state information from the activities executed before and after a given state. Based on this, there are basically three approaches to defining the state of a partially executed case in a log:

- *past*, i.e., the state is constructed based on the history of a case,
- *future*, i.e., the state of a case is based on its future, or
- *past and future*, i.e., a combination of the previous two.

Figure 21 shows an *example* of a trace and the three different "ingredients" that can be used to calculate state information. Given a concrete trace, i.e., the execution of a case from beginning to end, we can look at the state after

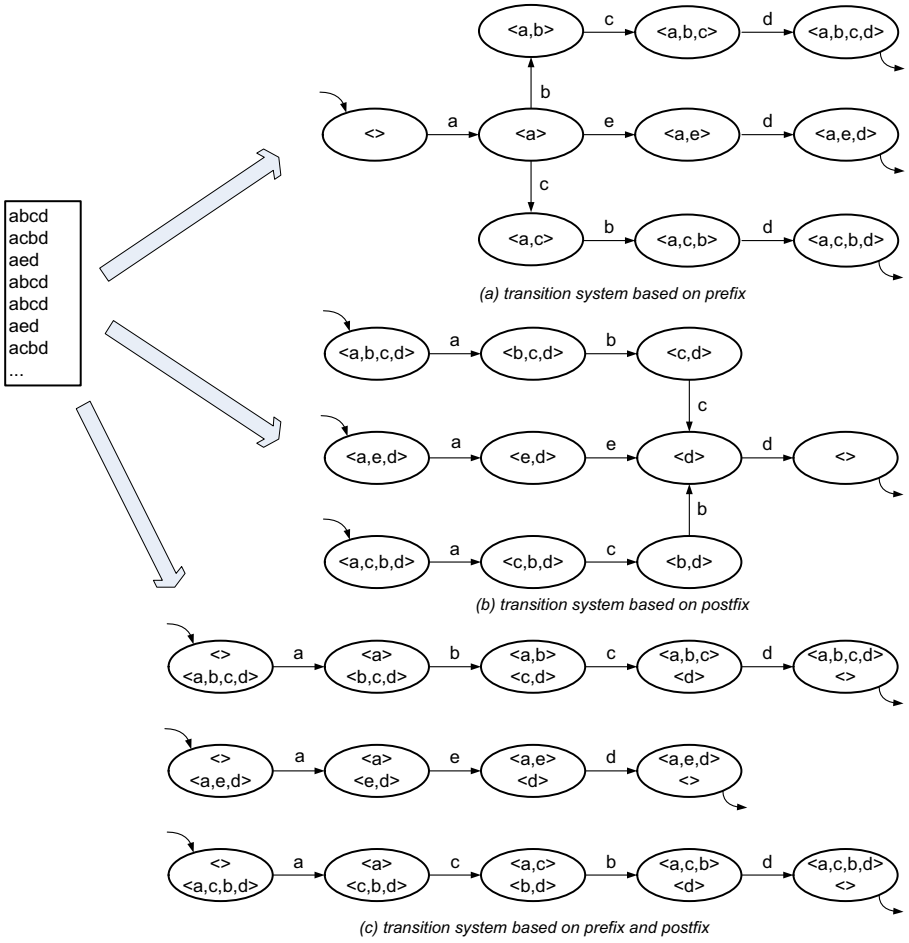


Fig. 22. Three transition systems derived from the log $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$

executing the first nine activities. This state can be represented by the prefix, the postfix, or both.

To explain the basic idea of constructing a transition system from an event log, consider Figure 22. If we just consider the prefix (i.e., the past), we get the transition system shown in Figure 22(a). Note that the initial state is denoted $\langle\rangle$, i.e., the empty sequence. Starting from this initial state the first activity is always a in each of the traces. Hence, there is one outgoing arc labeled a, and the subsequent state is labeled $\langle a\rangle$. From this state, three transitions are possible all resulting in different states, e.g., executing activity b results in state $\langle a, b\rangle$, etc. Note that in Figure 22(a) there is one initial state and three final states. Figure 22(b) shows the transition system based on postfixes. Here the state of a case is determined by its future. This future is known because process mining looks at the event log containing completed cases. Now there are three initial states and one final state. Initial state $\langle a, e, d\rangle$ indicates that the next activity will be a, followed by e and d. Note that the final state has label $\langle\rangle$ indicating that no activities need to be executed. Figure 22(c) shows a transition system based on both past and future. The node with label "$\langle a, b\rangle$,$\langle c, d\rangle$" denotes the state where a and b have happened and c and d still need to occur. Note that now there are three initial states and three final states.

The past of a case is a prefix of the complete trace. Similarly, the future of a case is a postfix of the complete trace. This may be taken into account completely, which leads to many different states and process models that may be too specific (i.e., "overfitting" models). It is also possible to take less information into account (e.g., just the last step in the process). This may result in "underfitting" models. The challenge is to select an abstraction that balances between "overfitting" and "underfitting". Many *abstractions* are possible; see for example the systematic treatment of abstractions in [8]. Here, we only highlight some of them.

Maximal horizon (h). The basis of the state calculation can be the complete prefix (postfix) or a *partial* prefix (postfix).

Filter (F). The second abstraction is to filter the (partial) prefix and/or postfix, i.e., activities in $F \subseteq A$ are kept while activities $A \setminus F$ are removed.

Maximum number of filtered events (m). The sequence resulting after filtering may contain a variable number of elements. Again one can determine a kind of horizon for this filtered sequence.

Sequence, bag, or set (q). The first three abstractions yield a sequence. The fourth abstraction mechanism optionally removes the order or frequency from the resulting trace.

Visible activities (V). The fifth abstraction is concerned with the transition labels. Activities in $V \subseteq A$ are shown explicitly on the arcs while the activities in $A \setminus V$ are not shown.

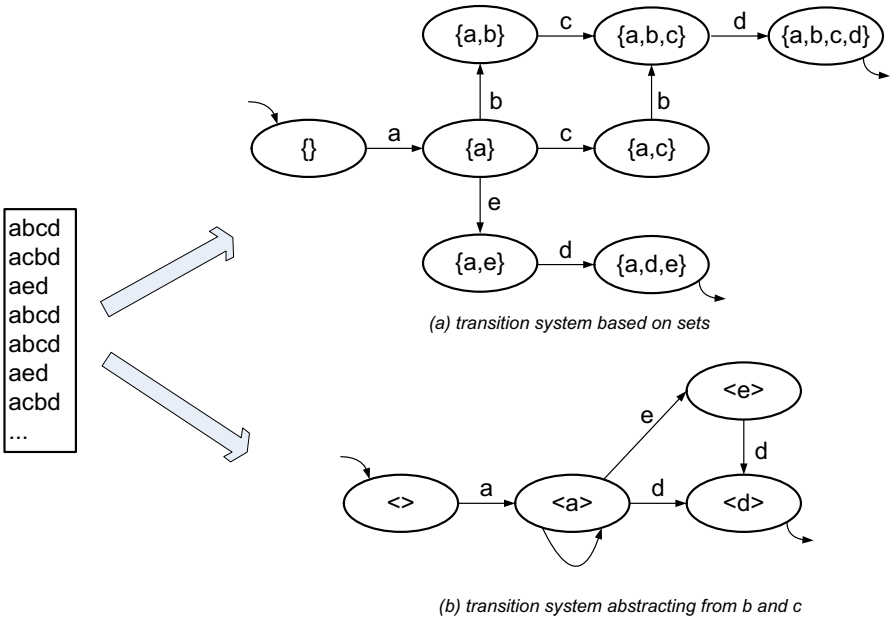


Fig. 23. Two transition systems built on L_1 using the following prefix abstractions: (a) $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = set$, and $V = A$, and (b) $h = \infty$, $F = \{a, d, e\}$, $m = 1$, $q = seq$, and $V = \{a, d, e\}$

Figure 23 illustrates the abstractions. In Figure 23(a) only the set abstraction is used $q = set$. The result is that several states are merged (compare with Figure 22(a)). In Figure 23(b) activities b and c are filtered out (i.e., $F = \{a, d, e\}$ and $V = \{a, d, e\}$). Moreover, only the last non-filtered event is considered for constructing the state (i.e., $m = 1$). Note that the states in Figure 23(b) refer to the last event in $\{a, d, e\}$. Therefore, there are four states: $\langle a \rangle$, $\langle d \rangle$, $\langle e \rangle$, and $\langle \rangle$. It is interesting to consider the role of b and c. First of all, they are not considered for building the state ($F = \{a, d, e\}$). Second, they are also not visualized ($V = \{a, d, e\}$), i.e., the labels are suppressed. The corresponding transitions are collapsed into the unlabeled arc from $\langle a \rangle$ to $\langle a \rangle$. If V would have included b and c, there would have been two such arcs labeled b respectively c.

The first four abstractions can be applied to the prefix, the postfix, or both. In fact, different abstractions can be applied to the prefix and postfix. As a result of these choices many different transitions systems can be generated. If more rigorous abstractions are used, the number of states will be smaller and the danger of "underfitting" is present. If, on the other hand, fewer abstractions are used, the number of states may be larger resulting in an "overfitting" model. An extreme case of overfitting was shown in Figure 22(c). At first this may seem confusing; however, as indicated in the introduction it is important to provide a repertoire of process discovery approaches. Depending on the desired degree of generalization, suitable abstractions are selected and in this way the analyst can

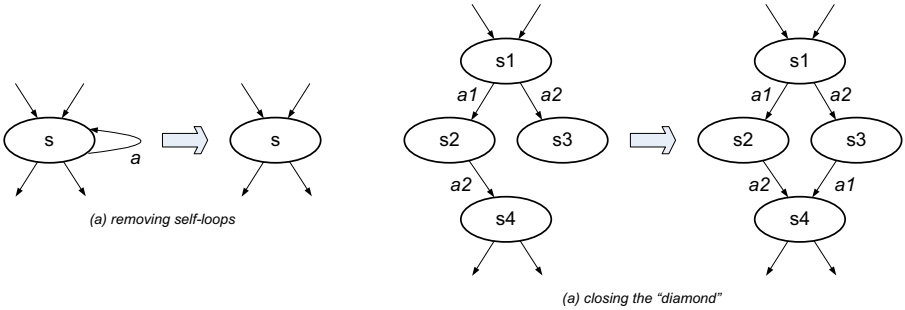


Fig. 24. Two examples of modifications of the transition system to aid the construction of the process model

balance between overfitting and underfitting, i.e., between generalization and precision in a controlled way.

Using classical region theory, we can transform the transition system into a process model. However, while we can now balance precision and generalization, we did not focus on simplicity yet. Therefore, we make use of the inner workings of state-based region theory to "massage" the transition system. This is intended to "pave the path" for region theory. For example, one may remove all "self-loops", i.e., transitions of the form $s \xrightarrow{a} s$ (cf. Figure 24(a)). The reason may be that one is not interested in events that do not change the state or that the synthesis algorithm in the second step cannot handle this. Another example would be to close all "diamonds" as shown in Figure 24(b). If $s_1 \xrightarrow{a_1} s_2$, $s_1 \xrightarrow{a_2} s_3$, and $s_2 \xrightarrow{a_2} s_4$, then $s_3 \xrightarrow{a_1} s_4$ is added. The reason for doing so may be that because (1) both a_1 and a_2 are enabled in s_1 and (2) after doing a_1, activity a_2 is still enabled, it is assumed that a_1 and a_2 can be executed in parallel. Although the sequence $\langle a_2, a_1 \rangle$ was not observed, it is assumed that this is possible and hence the transition system is extended by adding $s_3 \xrightarrow{a_1} s_4$.

7.2 From Transition Systems to Petri Nets

In the second step, the transition system is transformed into a Petri net using the techniques described in [13,15,23,24,26,27,35]. In Section 6.1, we introduced the basic idea of state-based regions. Therefore, we do not elaborate on this here. The important thing to note is that there is range of techniques to convert a transition system into a Petri net. These techniques typically only address two of the four quality dimensions mentioned in Figure 15: *fitness* and *precision*. The other two dimensions—*simplicity* and *generalization*—need to be addressed when constructing the transition system or by imposing additional constraints on the Petri net.

The goal of process mining is to present a model that can be interpreted easily by process analysts and end-users. Therefore, complex patterns should be avoided. Region-based approaches have a tendency to introduce "smart places",

i.e., places that compactly serve multiple purposes. Such places have many connections and may have non-local effects (i.e., the same place is used for different purposes in different phases of the process). Therefore, it may be useful to guide the generation of places such that they are easier to understand. This is fairly straightforward in both state-based region theory and language-based region theory. In [26, 27] it is shown that additional requirements can be added with respect to the properties of the resulting net. For example, the net can be forced to be free-choice, pure, etc. See [8] for examples.

The approach was already illustrated using Figure 18. Figure 25 shows some more examples based on the transition systems in figures 22 and 23. These models where computed using the classical synthesis approach presented in [26, 27]. This approach applies label-splitting if needed. Note that all transition systems were derived from event log $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$. The Petri net in Figure 25(a) is obtained by applying state-based region theory to the transition system in Figure 22(a). The same model is obtained when computing the regions for the transition system in Figure 23(a). The Petri net in Figure 25(b) is obtained when applying state-based region theory to the transition system in Figure 22(b). Two things can be noted: (1) the multiple initial states in Figure 22(b) result in many initial tokens and source places, and (2) label splitting

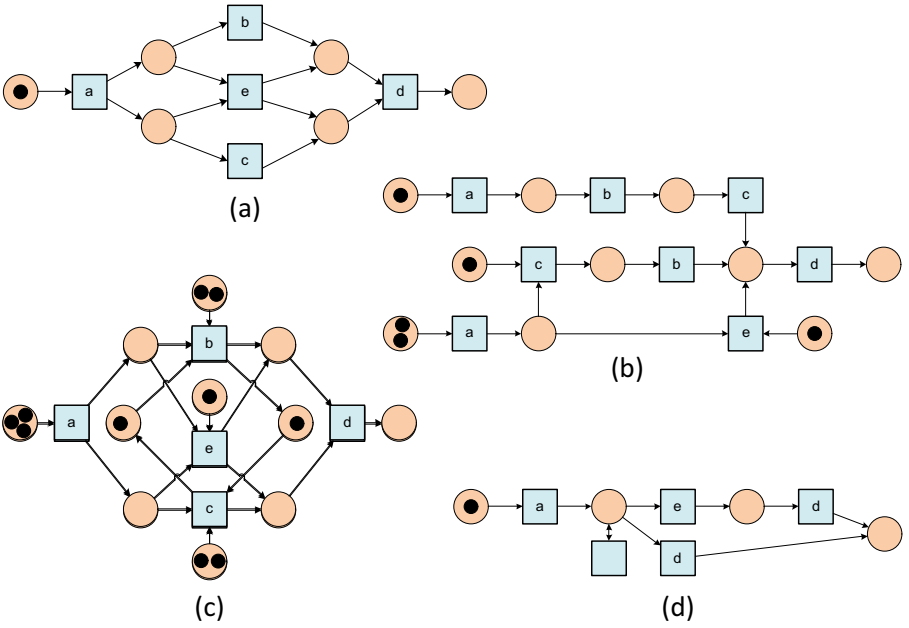


Fig. 25. Various Petri nets derived for the transitions systems in figures 22 and 23 using state-based regions. All models are based on event log $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$.

is used (e.g., there are two a transitions) to allow for the multiple starting points. The region-based approach synthesizes the model in Figure 25(c) for the transition system in Figure 22(c). Also this model suffers from the problem that there are multiple initial states. In general, we suggest to avoid having multiple initial states in the transition system to be synthesized. It is trivial to merge the initial states or add a new artificial initial state before applying region-based synthesis. Figure 25(d) was obtained from the transition system in Figure 23(b). The Petri net shows that if we abstract from b and c, we obtain an unlabeled transition indicating the state in which b and c would have occurred. This silent transition is due to the self-loop in the transition system of Figure 23(b). Eliminating the self-loop using the strategy presented in Figure 24(a) would remove the unlabeled transition in Figure 25(d).

7.3 Challenges

In this section, we have shown that by combining abstraction techniques and region theory, a powerful process mining algorithm can be obtained. Through several abstractions, we can obtain the desired level of precision and generalization, while by massaging the transition system, we can try to obtain simple models. However, there are also some drawbacks of this approach.

It is far from trivial to select the "right" parameters for the abstractions. Existing techniques and tools are sensitive to changes of parameter values, and the result is often unpredictable. Hence, obtaining a suitable process model is a matter of trial-and-error. Figure 26 shows, for example, the settings with which we can obtain the desired model for log L_6, i.e., the Petri net with a self-loop on transition b. However, the model shown in Figure 27 illustrates that the wrong settings may lead to an overfitting model.

Nonetheless, the state-based region approach is one of the few that can detect long-term dependencies, as shown by Figure 28, which resulted from applying the technique to log L_8.

Furthermore, the major drawback of the approach outlined here is the computational complexity. For larger logs, the resulting transition system may not fit in main memory and second, the region theory used to obtain a Petri net has a time complexity which is exponential in the size of the transition system.

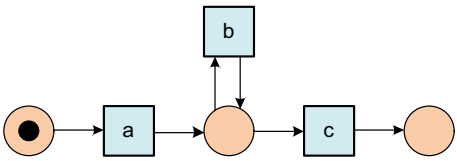


Fig. 26. Petri net obtained using region theory applied to log $L_6 = [\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2]$ using the following settings: $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = set$, and $V = A$ and a post-processing step in which states with identical inflow or outflow are merged

8 Process Discovery Using Integer Linear Programming

In Section 7, we have presented a two-step approach to apply region theory in the context of process mining. We focussed on obtaining a transition system from an event log and used classical region theory to obtain a Petri net. In this section, we do not consider the region theory as a black box, but instead, we extend existing approaches to make them more applicable in the context of process discovery, mainly by allowing the techniques to generalize from the log and to produce simpler models.

Both a basis and the separating representations of regions presented in [16,43] are based on the same principle, namely that a finite representation is provided of the infinite set of places satisfying Definition 11. By doing so, the language-based region theory ensures maximal preciseness and fitness, with little to no generalization and no aim for simple models. Hence, only two of the four quality dimensions of Figure 15 are considered.

For process discovery, we are aiming at simple, generalizing models. Hence, we present an approach [60], where we only represent those places satisfying Definition 11 that:

- each place expresses a causal dependency clearly visible in the log,
- no implicit places are included in the net, and
- places which are more expressive than others are favored, i.e., places with minimal input transitions and maximal output transitions are favored.

In contrast to the state-based region approach, we do not try to influence generalization and precision directly. Instead, we focus on model simplicity, by limiting the number of places in the model (and allowing for varying this number). As with the state-based approach, maximal fitness is guaranteed. In order to select places satisfying Definition 11, we convert this equation into a Integer Linear Programming (ILP) problem.

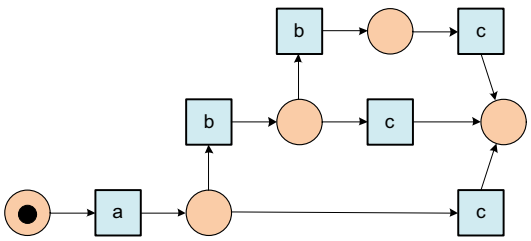


Fig. 27. Petri net obtained using region theory applied to log L_6 using the following parameters: $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = multiset$, and $V = A$

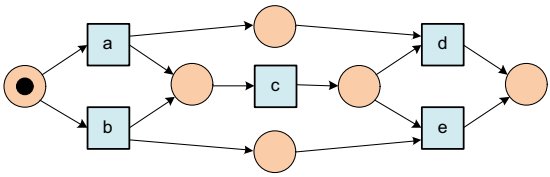


Fig. 28. Petri net obtained using region theory applied to log $L_8 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$ using the following settings: $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = set$, and $V = A$

8.1 Integer Linear Programming Representation

We quantify the expressiveness of places, in order to provide a target function, necessary to translate the inequation system of Definition 11 into an *Integer Linear Programming* (ILP) problem. In Section 8.2, we then use the result to generate a Petri net in a step-by-step fashion. In Section 8.3, we provide insights into the causal dependencies found in a log and how these can be used for finding places.

To apply the language-based theory of regions in the field of process discovery, we need to represent the event log as a prefix-closed language, i.e., by all the traces present in the event log, and their prefixes. Recall from Definition 1 that an event log is a finite bag of traces.

Definition 13 (Language of an event log). *Let A be a set of activities. Let $L \in \mathbb{B}(A^*)$ be an event log over this set of activities. The language $\mathcal{L}$ that represents this event log, uses alphabet A, and is defined by:*

$$\mathcal{L} = \{\sigma \in A^* \mid \exists \sigma' \in L \colon \sigma \leq \sigma'\}$$

A trivial Petri net capable of reproducing a language is a net with only transitions. This net is simple, can represent all traces, and hence has maximal fitness. It also generalizes well, but the Petri net with only transitions is very imprecise because anything is possible according to the model. To restrict the behavior allowed by the Petri net, but not observed in the log, we start adding places to that Petri net. As stated before, the places we add to the Petri net should be *as expressive as possible*, which is the same as saying that such places have a maximal postset and a minimal preset, i.e., it should not be possible to add an output transition to or to remove an input transition from a place without reducing the fitness of the resulting net.

Besides searching for regions that lead to places with maximum expressiveness, we also want to avoid adding implicit places to a model. Therefore, we will search for "minimal regions" as introduced in [30]. Using the inequation system of Definition 11 and the expressiveness of a place, we can define a target function for our ILP problem to construct the places of a Petri net in a logical order [52].

The following target function is shown to be such that it favors minimal regions which are maximally expressive [60]:

Definition 14 (Target function). *Let A be a set of activities. Let $L \in \mathcal{B}(A^*)$ be an event log and $\mathcal{L}$ the corresponding language. Furthermore, let M be the matrix defined in Definition 11. We define the function $\tau : \Re(\mathcal{L}) \to \mathbb{N}$ by*

$$\tau((\vec{x}, \vec{y}, c)) = c + \vec{1}^{\,T}(\vec{1} \cdot c + M \cdot (\vec{x} - \vec{y}))$$

Combining this target function with the inequation system of Definition 11 yields the following ILP problem:

Definition 15 (ILP formulation). *Let A be a set of activities, let $L \in \mathcal{B}(A^*)$ be an event log, and let M and M' be the matrices as defined in Definition 11. We define the ILP ILP_L for event log L as:*

$$\begin{array}{lrll}
\textit{Min } c + \vec{1}^{\,T}(\vec{1} \cdot c + M \cdot (\vec{x} - \vec{y})) & & & \textit{Definition 14} \\
\textit{s.t.} & c \cdot \vec{1} + M' \cdot \vec{x} - M \cdot \vec{y} \geq \vec{0} & & \textit{Definition 11} \\
& \vec{1}^{\,T} \cdot \vec{x} + \vec{1}^{\,T} \cdot \vec{y} \geq 1 & & \textit{There should be at least one edge} \\
& \vec{0} \leq \vec{x} \leq \vec{1} & x \in \{0,1\}^{|T|} & \\
& \vec{0} \leq \vec{y} \leq \vec{1} & y \in \{0,1\}^{|T|} & \\
& 0 \leq c \leq 1 & c \in \{0,1\} &
\end{array}$$

This ILP problem provides the basis for our process discovery problem. However, an optimal solution to this ILP only provides a single feasible place with a minimal value for the target function. Therefore, in the next section, we show how this ILP problem can be used as a basis for constructing a Petri net from a log.

8.2 Constructing Petri Nets Using ILP

In the previous subsection, we provided the basis for adding places to a Petri net based on knowledge extracted from a log. In fact, the target function of Definition 14 provides a partial order on all elements of the set $\Re(\mathcal{L})$, i.e., the set of all regions of a language. In this subsection, we show how to generate the first n places of a Petri net, that is (1) able to reproduce a log under consideration and (2) of which the places are as expressive as possible.

A trivial approach would be to add each found solution as a negative example to the ILP problem, i.e., explicitly forbidding this solution. However, it is clear that once a region r has been found and the corresponding feasible place is added to the Petri net, we are no longer interested in regions r' for which the corresponding feasible place has more tokens, less outgoing arcs or more incoming arcs, i.e., we are only interested in independent regions.

Definition 16 (Refining the ILP after each solution). *Let A be a set of activities, let $L \in \mathcal{B}(A^*)$ be an event log, let M and M' be the matrices as defined in Definition 11 and let ILP_L^0 be the corresponding ILP. Furthermore, for $i \geq 0$ let region $r_i = (\vec{x_i}, \vec{y_i}, c_i)$ be a minimal solution of ILP_L^i. We define the refined ILP as ILP_L^i, with the extra constraint specifying that:*

$$-c_i \cdot c + \vec{y}^{\,T} \cdot (\vec{1} - \vec{y_i}) - \vec{x}^{\,T} \cdot \vec{x_i} \geq -c_i + 1 - \vec{1}^{\,T} \cdot \vec{x_i}$$

Note that for any solution $r = (\vec{x}, \vec{y}, c)$ of ILP^i_L: $c < c_i$ or there is a $t \in A$ such that $\vec{x}(t) < \vec{x}_i(t)$ or $\vec{y}(t) > \vec{y}_i(t)$. If this is not the case (i.e., $c \geq c_i$ and $\vec{x}(t) \geq \vec{x}_i(t)$ and $\vec{y}(t) \leq \vec{y}_i(t)$ for any t), then $-c_i \cdot c = -c_i$, $-\vec{x}(t) \cdot \vec{x_i}(t) = -x_i(t)$, and $\vec{y}(t) \cdot \vec{y_i}(t) = 0 \not\geq 1$. Hence, we find a contradiction with respect to the extra constraint. As a result the new region r is forced to be sufficiently different from r_i.

The refinement operator presented above, basically defines an algorithm for constructing the places of a Petri net that is capable of reproducing a given log. The places are generated in an order which ensures that the most expressive places are found first and that only places are added that have less tokens, less outgoing arcs, or more incoming arcs. Furthermore, each solution of a refined ILP is also a solution of the original ILP, since the new solution satisfies all constraints of the initial ILP formulation, and some extra constraints. Hence, all places constructed using this procedure are feasible places.

This procedure, can be used to continue adding places, thus making the model more precise, while compromising on model complexity as shown by Figure 29. The Petri net in Figure 29 allows for more behavior than the log L_1 contains, so in theory more places could still be added. Nonetheless, any new place would be such that it has fewer output arcs, or more input arcs than the ones included in this model. In the worst case, the total number of places introduced is exponential in the number of transitions. Since there is no way to provide insights into an upperbound for the number of places to generate, we propose a more suitable approach, *not using the refinement step of Definition 16*. Instead, we propose to guide the search for solutions (i.e. for places) using concepts from the α-algorithm [9].

8.3 Using Log-Based Properties

Recall from the beginning of this section, that we are specifically interested in places expressing explicit causal dependencies between transitions. In this

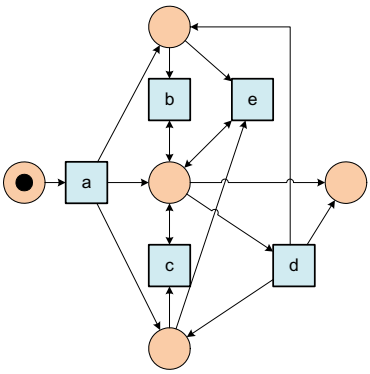


Fig. 29. Petri net obtained using language-based region theory naively applied to log L_1

subsection, we use the causal relations $\rightarrow_L$ defined earlier in Definition 7 in combination with the ILP of Definition 15 to construct a Petri net.

Causal dependencies between transitions are used by many process discovery algorithms [6,9,31,58] and generally provide a good indication as to which transitions should be connected through places. Furthermore, extensive techniques are available to derive causal dependencies between transitions using heuristic approaches [9, 31]. However, it is not known whether the log is complete and whether we covered all causal dependencies. Therefore, we restrict ourselves to search for a Petri net such that if a causal dependency is not in the log, it is also not in the net. In order to find a place expressing a specific causal dependency, we extend the ILP presented in Definition 15.

Definition 17 (ILP for causal dependency). *Let A be a set of activities, let $L \in \mathbb{B}(A^*)$ be an event log, let M and M' be the matrices as defined in Definition 11 and let ILP_L be the corresponding ILP. Furthermore, let $t_1, t_2 \in A$ and assume $t_1 \rightarrow_L t_2$. We define the refined ILP, $ILP_L^{t_1 \rightarrow t_2}$ as ILP_L, with two extra bounds specifying that:*

$$\vec{x}(t_1) = \vec{y}(t_2) = 1$$

A solution of the optimization problem expresses the causal dependency $t_1 \rightarrow_L t_2$, and restricts the behavior as much as possible. However, such a solution does not have to exist, i.e., the ILP might be infeasible, in which case no place is added to the Petri net being constructed. Nonetheless, by considering a separate ILP for each causal dependency in the log, a Petri net can be constructed, in which each place is as expressive as possible and expresses at least one dependency derived from the log. With this approach, at most one place is generated for each dependency and thus the upper bound of places in $N(\mathcal{L})$ is the number of causal dependencies, which is worst-case quadratic in the number of transitions.

The result of applying this log-based technique to our log L_1 is shown in Figure 30. This model is very close to the desired model, except that it does not contain a final place. This is a general drawback of language-based region theory: the focus is on the ability to reproduce prefixes of log traces rather than termination in a well-defined final state.

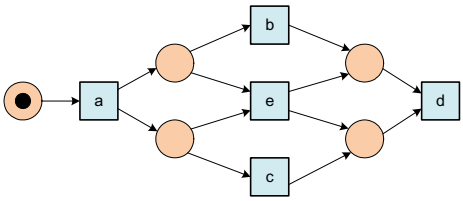


Fig. 30. Petri net obtained using language-based region theory using log-based properties applied to log L_1. Note that compared to earlier solutions the sink place denoting termination is missing.

Up to now, we did not impose any restriction on the structure of the resulting Petri net. By adding constraints, several Petri net properties can be expressed, thus resulting in elementary nets, pure nets, (extended) free-choice nets, state machines and marked graphs [60]. This allows us to further simplify the resulting Petri net. Note that this is similar to the refinement described in Section 7.2 for state-based regions.

8.4 Challenges

In sections 7 and 8 we presented several ways to use region theory in the context of process discovery in order to alleviate some of the problems of the α-algorithm. First, we have shown how to we can balance precision and generalization while constructing a transition system from a log. Then, by massaging the transition system, we can somewhat improve the simplicity of the resulting models. When using language-based region theory, we have shown that we can focus on the simplicity of the resulting model. By incrementally introducing places, we can make the resulting model more precise in a step-by-step fashion. Figures 31 and 32 show that we can discover models for the logs L_6 and L_8, but the long-term dependency in L_8 is not identified, due to the reliance on the causal dependencies used in the α-algorithm. Furthermore, as discussed before, language-based regions have problems making the final state explicit (i.e., sink places are missing in figures 31 and 32).

Unfortunately, all region-based approaches are computationally challenging. In the case of the language-based regions, finding a solution for each incremental ILP problem is of worst-case exponential time complexity. Furthermore, the

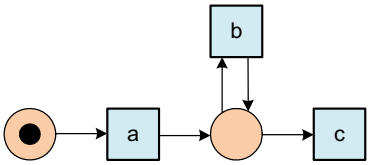


Fig. 31. Petri net discovered for event log L_6. The model was obtained using language-based region theory guided by log-based properties.

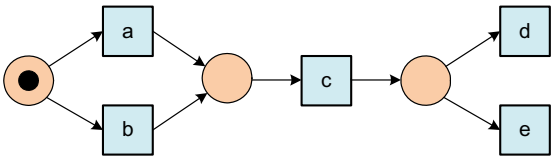


Fig. 32. Petri net obtained using language-based region theory guided by log-based properties applied to log $L_8 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$. No sink place is created and the long-term dependencies are not discovered because only short-term dependencies are used to guide the discovery of places.

common property of all region-based techniques is that the fitness of the discovered net is guaranteed to be 100%, regardless of the log. This makes these approaches very robust, but also sensitive to noise.

Thus far we only used toy examples to illustrate the different concepts. All functionality has been embedded in the process discovery framework ProM, which is capable of constructing nets for logs with thousands of cases referring to dozens of transitions. The techniques have been tested on many real-life and synthetic event logs. However, a discussion of these experimental results is outside the scope of this article. For this we refer to [2, 7, 39, 47, 53].

9 Tool Support

Both for process mining and region theory, it is essential that algorithms can be put to the test in real life environments. Therefore, almost all work presented in this article is implemented in freely available tools. For example, classical state-based region theory is implemented in Petrify and Genet [22], while Rbminer [54] applies this in a process discovery context. Some of the language-based region theory is implemented in VIPTool [18].

The process mining algorithms presented in sections 4, 7 and 8 have all been implemented in the *ProM framework* [11,56,57]. All algorithms discussed in this article can be found the most recent version of ProM (version 6.0 and later). ProM is a generic open-source framework for implementing process mining algorithms in a standard environment.

Figure 33 shows the startup screen of ProM. Here, a log was opened for analysis which is shown in the workspace. When selecting the log and clicking on the action button, the user is taken to the action browser, where in Figure 34, the α-miner is selected. The α-miner is an implementation of the work in Section 4.

In earlier versions of ProM, the actual process mining algorithms implemented by plug-ins assumed the presence of a GUI. Most algorithms require parameters,

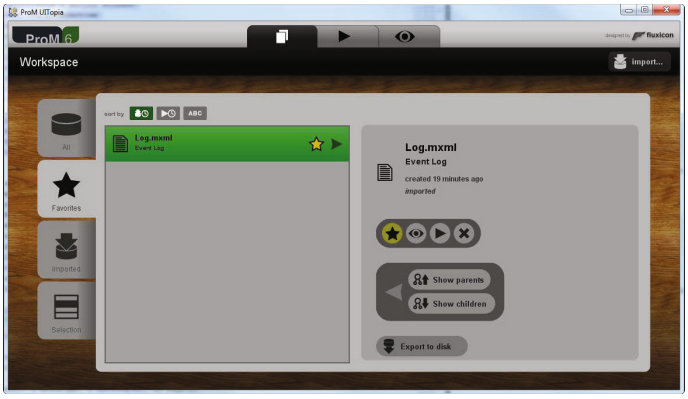


Fig. 33. ProM 6 Workspace; opening screen after loading a file

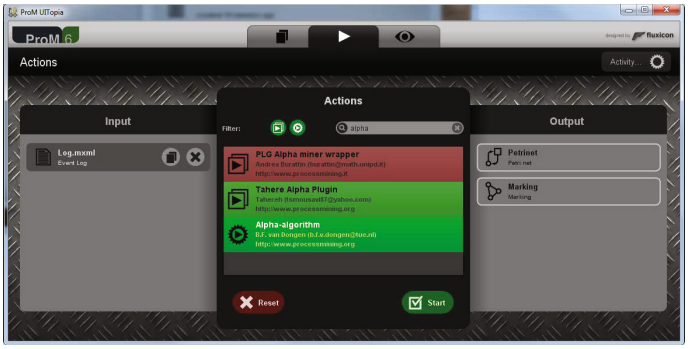


Fig. 34. ProM 6 Action Browser; selecting the *alpha*-miner to discover a process model from the loaded event log

and the plug-in would ask the user for these parameters using some GUI-based dialog. Furthermore, some plug-ins displayed status information using progress bars and such. Thus, the actual process mining algorithm and the use of the GUI were intertwined. As a result, the algorithm could only be run in a GUI-aware context, say on a local workstation. This way, it was impossible to effectively run process mining experiments using a distributed infrastructure and/or in batch.

In ProM 6, the process mining algorithm and the GUI have been carefully separated, and the concepts of *contexts* has been introduced. For a plug-in, the context is the proxy for its environment, and the context determines what the plug-in can do in its environment. A plug-in can only display a dialog or a progress bar on the display if the context is GUI-aware. Typically, in ProM 6, the implementation of an algorithm is split into a number of plug-ins: A plug-in for every context. The actual process mining algorithm will be implemented in a generic way, such that it can run in a general (GUI-unaware) context. This allows the algorithm to be run in any context, even in a distributed context [20]. The dialog for setting the required parameters is typically implemented in a GUI-aware variant of the plug-in. Typically, this GUI-aware plug-in first displays the parameter dialog, and when the user has provided the parameters and has closed the dialog, it will simply run the generic plug-in using the provided parameters.

The major advantage of this is that the ProM framework may decide to have the generic plug-in run on a different computer than the local workstation. Some plug-ins may require lots of system resources (e.g., computing power, memory, and disk space), like for example the genetic miner. Basically, the genetic miner takes a model and a log, and then generates a number of alternative models for the given log. The best of these alternative models are then taken as new starting points for the genetic miner. The genetic miner repeats this until some stop criterion has been reached, after which it returns the best model found so far. Clearly, this miner might take considerable time (it may take hundreds of iterations before it stops and the fitness calculation is very time-consuming for large logs), and it may take considerable memory (the number of alternative

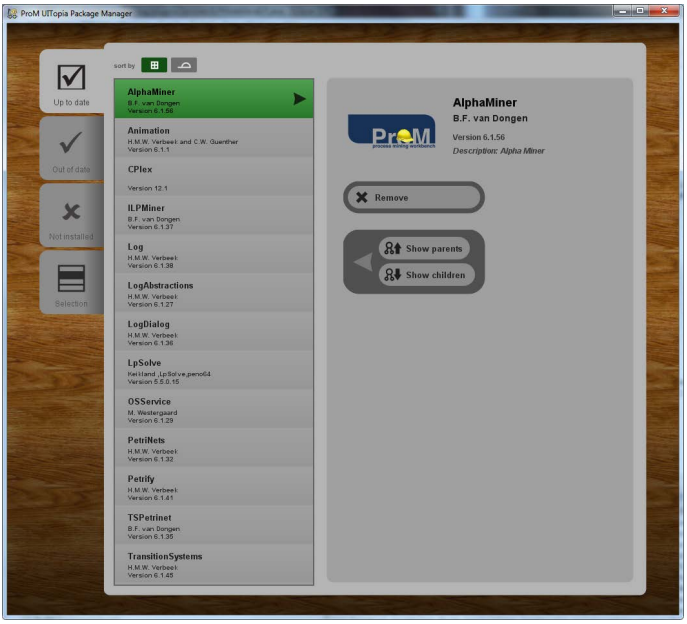


Fig. 35. ProM 6 Package manager showing the packages relevant for the techniques presented in this article

models may grow rapidly). For such an algorithm, it might be preferable to have it run on a server which is more powerful than the local workstation. Moreover, genetic mining can be distributed in several ways [20]. For example, the population can be partitioned over various nodes. Each subpopulation on a node evolves independently for some time after which the nodes exchange individuals. Similarly, the event logs may be portioned over nodes thus speeding up the fitness calculations.

Besides separating the functionality from the user interface, ProM 6 requires functionality to be provided in *packages*. These packages each contain a collection of related algorithms, typically implemented by one research group. When ProM is started for the first time, the package manager is opened as shown in Figure 35. Here, for each known package, ProM shows who the author is, what the current version is and whether or not this version is installed. The work presented in this article, requires the following packages to be installed: `AlphaMiner`, `TransitionSystems` and `ILPMiner`. The other packages shown are automatically installed due to dependencies. Furthermore, the package `Petrify` provides import and export functionality to and from the state-based region tool Petrify.

The event log opened in Figure 33 is a log consisting of 1000 cases of a travel agency. A customer registers, then purchases a bus ticket or a plane ticket while at the same time he books one or more hotels. After the booking phase, the trip costs are computed and the customer has to choose between two types of

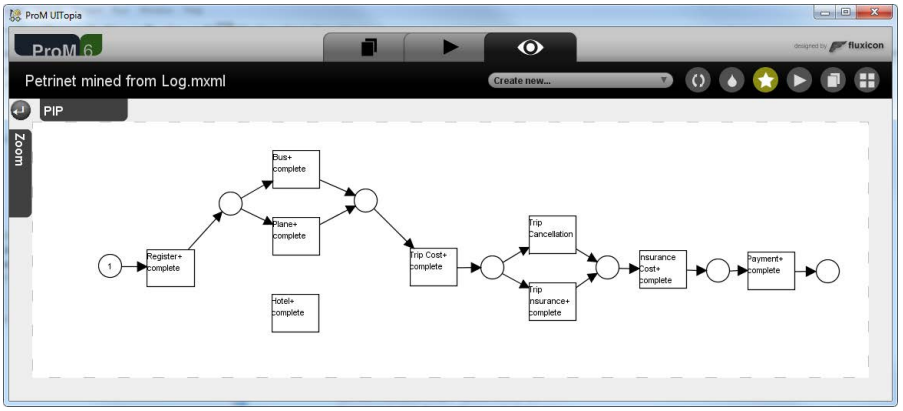


Fig. 36. Result of α-miner: the α-algorithm has problems dealing with the multiple hotel bookings interleaved with other booking activities

insurance. After that, the total costs are calculated and the payment is completed. This is a rather simple example used to show the results of the three algorithms.

The resulting Petri net after applying the α-algorithm to this log is shown in Figure 36. The result after executing the transition system miner is shown in Figure 37 and the result of the ILP miner is shown in 38. All three algorithms provide a model that indeed models the given situation. The difficulty here is the fact that the hotel booking is executed one or more times. The α-algorithm does not connect this transition (thus enabling it continuously and destroying the WF-net structure), while the transition system miner introduces two transitions for this step, but it enforces that the second hotel can only be booked after the

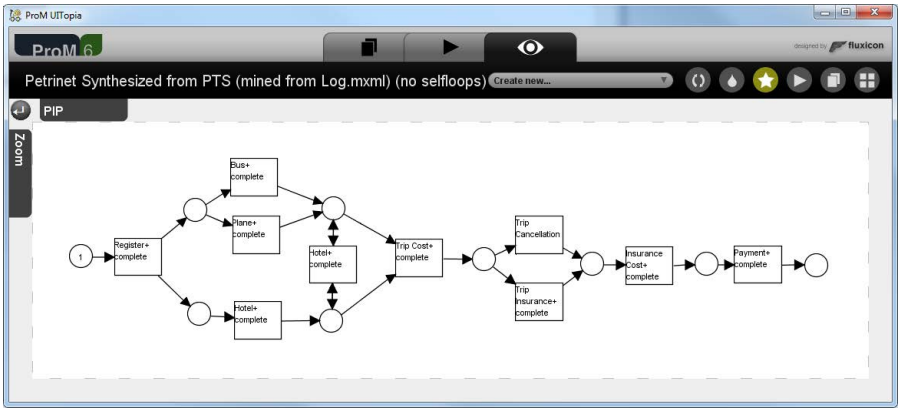


Fig. 37. Result of TS Miner. Note that there are now two transitions referring to hotel bookings (label splitting).

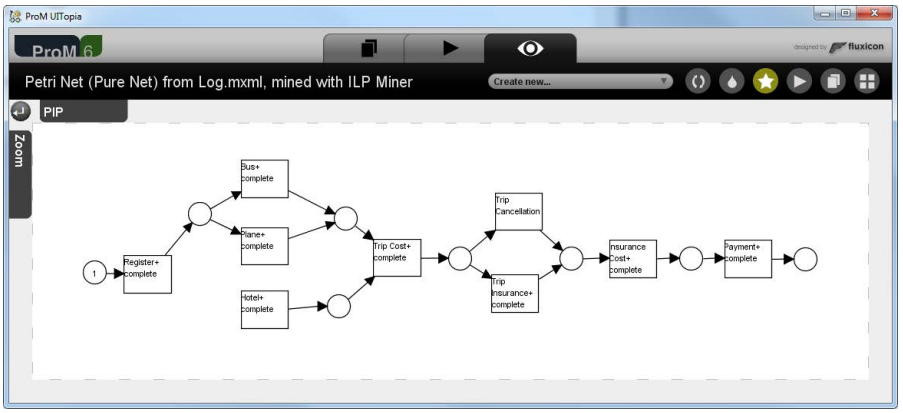


Fig. 38. Result of ILP Miner. The model is able to replay the event log. However, tokens may remain in the place following the hotel booking and bookings can take place before the registration step.

bus or plane ticket is booked. The ILP miner allows for the hotel booking to occur arbitrarily often, but at least once before the trip costs are calculated.

10 Conclusion

Process mining can be seen as the "missing link" between data mining and traditional model-driven BPM. The spectacular growth of event data is an important enabler for process analysis based on real observations rather than hand-made models only. We have applied ProM in over 100 organizations ranging from municipalities and hospitals to financial institutions and manufacturers of high-tech systems. This illustrates the applicability of the techniques described in this article.

Process mining can be used to diagnose the actual processes. This is valuable because in many organizations most stakeholders lack a correct, objective, and accurate view on important operational processes. However, process mining is not limited to the process discovery techniques mentioned in this article (see for example [2]). Process mining can also be used to improve the discovered processes. Conformance checking can be used for auditing and compliance. By replaying the event log on a process model it is possible to quantify and visualize deviations. Similar techniques can be used to detect bottlenecks and build predictive models. Given the applicability of process mining, we encourage the reader to simply apply the techniques discussed. The event data needed to conduct such experiments can be found in any non-trivial organization. The freely available open-source process mining tool ProM can be downloaded from www.processmining.org and supports all of the process mining techniques mentioned.

In this article we emphasized that four quality dimensions—*fitness*, *simplicity*, *precision*, and *generalization*—need to be balanced [2]. Moreover, we zoomed

in on region-based approaches. As shown, conventional *state-based regions* and *language-based regions* focus on fitness and precision, while neglecting simplicity and generalization. Fortunately, it is possible to modify these techniques to also deal with the other two quality dimensions. State-based regions can be used for process discovery tasks provided that the right abstraction is used when constructing the transition system. Language-based regions can be mapped onto an ILP problem where the target function and additional constraints are used to obtain a simple and more general model.

Despite the applicability of process mining there are many interesting challenges; these illustrate that process mining is a young discipline. As discussed, it is far from trivial to construct a process model based on event logs that are incomplete and noisy. Unfortunately, there are still researchers and tool vendors that assume logs to be complete and free of noise. Although heuristic mining, genetic mining, and fuzzy mining provide case-hardened process discovery techniques, many improvements are needed to construct truly intuitive models that are able to explain the most likely/common behavior. Another challenge is to deal with ever-growing datasets, i.e., it is not uncommon to have event logs with millions of cases, billions of events, and thousands of activities [44]. In some cases it is impossible to store all events and process models need to be discovered on-the-fly. In other cases, there is a need to distribute process mining problems over multiple computers. As discussed in [3] this can be done in various ways. Therefore, there are many interesting problems for researchers with a background in Petri nets and eager to analyze processes based on real event data rather than unrealistic toy models.

Acknowledgments. The authors would like to thank all the people that contributed to the development of ProM (www.processmining.org).

References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Berlin (2011)
3. van der Aalst, W.M.P.: Distributed Process Discovery and Conformance Checking. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 1–25. Springer, Heidelberg (2012)
4. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Replaying History on Process Models for Conformance Checking and Performance Analysis. WIREs Data Mining and Knowledge Discovery 2(2), 182–192 (2012)
5. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., de Medeiros, A.K.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W., Weijters, A.J.M.M.: ProM 4.0: Comprehensive Support for *Real* Process Analysis. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 484–494. Springer, Heidelberg (2007)

6. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow Mining: A Survey of Issues and Approaches. Data and Knowledge Engineering 47(2), 237–267 (2003)
7. van der Aalst, W.M.P., Reijers, H.A., Weijters, A.J.M.M., van Dongen, B.F., Alves de Medeiros, A.K., Song, M., Verbeek, H.M.W.: Business Process Mining: An Industrial Application. Information Systems 32(5), 713–732 (2007)
8. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. Software and Systems Modeling 9(1), 87–111 (2010)
9. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
10. Agrawal, R., Gunopulos, D., Leymann, F.: Mining Process Models from Workflow Logs. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 469–483. Springer, Heidelberg (1998)
11. van der Aalst, W.M.P., van Dongen, B., Günther, C.W., Rozinat, A., Verbeek, E., Weijters, T.: ProM: The Process Mining Toolkit. In: de Medeiros, A.K.A., Weber, B. (eds.) Business Process Management Demonstration Track (BPMDemos 2009). CEUR Workshop Proceedings, vol. 489, pp. 1–4. CEUR-WS.org (2009)
12. Angluin, D., Smith, C.H.: Inductive Inference: Theory and Methods. Computing Surveys 15(3), 237–269 (1983)
13. Badouel, E., Bernardinello, L., Darondeau, P.: The Synthesis Problem for Elementary Net Systems is NP-complete. Theoretical Computer Science 186(1-2), 107–134 (1997)
14. Badouel, E., Bernardinello, L., Darondeau, P.: Polynomial Algorithms for the Synthesis of Bounded Nets. In: Mosses, P.D., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 364–378. Springer, Heidelberg (1995)
15. Badouel, E., Darondeau, P.: Theory of Regions. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
16. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007)
17. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of Petri Nets from Infinite Partial Languages. In: International Conference on Application of Concurrency to System Design (ACSD 2008), pp. 170–179. IEEE Computer Society (2008)
18. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of Petri Nets from Scenarios with VipTool. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 388–398. Springer, Heidelberg (2008)
19. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. Fundamenta Informaticae 95(1), 187–217 (2009)
20. Bratosin, C., Sidorova, N., van der Aalst, W.M.P.: Distributed Genetic Process Mining. In: Ishibuchi, H. (ed.) IEEE World Congress on Computational Intelligence (WCCI 2010), Barcelona, Spain, pp. 1951–1958. IEEE (July 2010)
21. Carmona, J., Cortadella, J., Kishinevsky, M.: A Region-Based Algorithm for Discovering Petri Nets from Event Logs. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 358–373. Springer, Heidelberg (2008)

22. Carmona, J., Cortadella, J., Kishinevsky, M.: Genet: A Tool for the Synthesis and Mining of Petri Nets. In: Application of Concurrency to System Design (ACSD 2009), pp. 181–185. IEEE Computer Society (2009)
23. Carmona, J., Cortadella, J., Kishinevsky, M.: New Region-Based Algorithms for Deriving Bounded Petri Nets. IEEE Transactions on Computers 59(3), 371–384 (2010)
24. Carmona, J., Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: A symbolic algorithm for the synthesis of bounded petri nets. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 92–111. Springer, Heidelberg (2008)
25. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology 7(3), 215–249 (1998)
26. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Synthesizing Petri Nets from State-Based Models. In: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1995), pp. 164–171. IEEE Computer Society (1995)
27. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri Nets from Finite Transition Systems. IEEE Transactions on Computers 47(8), 859–882 (1998)
28. Darondeau, P.: Deriving Unbounded Petri Nets from Formal Languages. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 533–548. Springer, Heidelberg (1998)
29. Datta, A.: Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. Information Systems Research 9(3), 275–301 (1998)
30. Desel, J., Reisig, W.: The Synthesis Problem of Petri Nets. Acta Informatica 33(4), 297–315 (1996)
31. van Dongen, B.F.: Process Mining and Verification. Phd thesis, Eindhoven University of Technology (2007)
32. van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase Process Mining: Building Instance Graphs. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 362–376. Springer, Heidelberg (2004)
33. van Dongen, B.F., van der Aalst, W.M.P.: Multi-Phase Mining: Aggregating Instances Graphs into EPCs and Petri Nets. In: Marinescu, D. (ed.) Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, pp. 35–58. Florida International University, Miami (2005)
34. van Dongen, B.F., Alves de Medeiros, A.K., Wen, L.: Process Mining: Overview and Outlook of Petri Net Discovery Algorithms. In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 225–242. Springer, Heidelberg (2009)
35. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures - Part 1 and Part 2. Acta Informatica 27(4), 315–368 (1989)
36. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM 43(3), 555–600 (1996)
37. Gold, E.M.: Language Identification in the Limit. Information and Control 10(5), 447–474 (1967)
38. Gold, E.M.: Complexity of Automaton Identification from Given Data. Information and Control 37(3), 302–320 (1978)

39. Günther, C.W., van der Aalst, W.M.P.: Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 328–343. Springer, Heidelberg (2007)
40. Herbst, J.: A Machine Learning Approach to Workflow Management. In: Lopez de Mantaras, R., Plaza, E. (eds.) ECML 2000. LNCS (LNAI), vol. 1810, pp. 183–194. Springer, Heidelberg (2000)
41. van der Aalst, W., Adriansyah, A., de Medeiros, A.K.A., Arcieri, F., Baier, T., Blickle, T., Bose, J.C., van den Brand, P., Brandtjen, R., Buijs, J., Burattin, A., Carmona, J., Castellanos, M., Claes, J., Cook, J., Costantini, N., Curbera, F., Damiani, E., de Leoni, M., Delias, P., van Dongen, B.F., Dumas, M., Dustdar, S., Fahland, D., Ferreira, D.R., Gaaloul, W., van Geffen, F., Goel, S., Günther, C., Guzzo, A., Harmon, P., ter Hofstede, A., Hoogland, J., Ingvaldsen, J.E., Kato, K., Kuhn, R., Kumar, A., La Rosa, M., Maggi, F., Malerba, D., Mans, R.S., Manuel, A., McCreesh, M., Mello, P., Mendling, J., Montali, M., Motahari-Nezhad, H.R., zur Muehlen, M., Munoz-Gama, J., Pontieri, L., Ribeiro, J., Rozinat, A., Seguel Pérez, H., Seguel Pérez, R., Sepúlveda, M., Sinur, J., Soffer, P., Song, M., Sperduti, A., Stilo, G., Stoel, C., Swenson, K., Talamo, M., Tan, W., Turner, C., Vanthienen, J., Varvaressos, G., Verbeek, E., Verdonk, M., Vigo, R., Wang, J., Weber, B., Weidlich, M., Weijters, T., Wen, L., Westergaard, M., Wynn, M.: Process Mining Manifesto. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM Workshops 2011, Part I. LNBIP, vol. 99, pp. 169–194. Springer, Heidelberg (2012)
42. Lorenz, R.: Towards Synthesis of Petri Nets from General Partial Languages. In: German Workshop on Algorithms and Tools for Petri Nets (AWPN 2008). CEUR Workshop Proceedings, vol. 380, pp. 55–62. CEUR-WS.org (2008)
43. Lorenz, R., Juhás, G.: How to Synthesize Nets from Languages: A Survey. In: Henderson, S.G., Biller, B., Hsieh, M., Shortle, J., Tew, J.D., Barton, R.R. (eds.) Proceedings of the Wintersimulation Conference (WSC 2007), pp. 637–647. IEEE Computer Society (2007)
44. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.: Big Data: The Next Frontier for Innovation, Competition, and Productivity. McKinsey Global Institute (2011)
45. Mauser, S., Lorenz, R.: Variants of the Language Based Synthesis Problem for Petri Nets. In: International Conference on Application of Concurrency to System Design (ACSD 2009), pp. 89–98. IEEE Computer Society (2009)
46. de Medeiros, A.K.A., van der Aalst, W.M.P., Weijters, A.J.M.M.T.: Workflow Mining: Current Status and Future Directions. In: Meersman, R., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 389–406. Springer, Heidelberg (2003)
47. de Medeiros, A.K.A., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
48. Munoz-Gama, J., Carmona, J.: Enhancing Precision in Process Conformance: Stability, Confidence and Severity. In: Chawla, N., King, I., Sperduti, A. (eds.) IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011), Paris, France. IEEE (April 2011)
49. Pitt, L.: Inductive Inference, DFAs, and Computational Complexity. In: Jantke, K.P. (ed.) AII 1989. LNCS, vol. 397, pp. 18–44. Springer, Heidelberg (1989)
50. Reisig, W., Rozenberg, G. (eds.): APN 1998. LNCS, vol. 1491. Springer, Heidelberg (1998)
51. Rozinat, A., van der Aalst, W.M.P.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems 33(1), 64–95 (2008)

52. Schrijver, A.: Theory of Linear and Integer programming. Wiley-Interscience (1986)
53. Solé, M., Carmona, J.: Process Mining from a Basis of State Regions. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 226–245. Springer, Heidelberg (2010)
54. Solé, M., Carmona, J.: Rbminer: A tool for discovering petri nets from transition systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 396–402. Springer, Heidelberg (2010)
55. Verbeek, H.M.W., Basten, T., van der Aalst, W.M.P.: Diagnosing Workflow Processes using Woflan. The Computer Journal 44(4), 246–279 (2001)
56. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: ProM 6: The Process Mining Toolkit. In: La Rosa, M. (ed.) Proc. of BPM Demonstration Track 2010. CEUR Workshop Proceedings, vol. 615, pp. 34–39 (2010)
57. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Soffer, P., Proper, E. (eds.) CAiSE Forum 2010. LNBIP, vol. 72, pp. 60–75. Springer, Heidelberg (2011)
58. Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. Integrated Computer-Aided Engineering 10(2), 151–162 (2003)
59. Wen, L., van der Aalst, W.M.P., Wang, J., Sun, J.: Mining Process Models with Non-Free-Choice Constructs. Data Mining and Knowledge Discovery 15(2), 145–180 (2007)
60. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundamenta Informaticae 94, 387–412 (2010)

Author Index